

IDC DOCUMENTATION

Database Tutorial



Notice

Every effort was made to ensure that the information in this document was accurate at the time of printing. However, the information is subject to change.

Contributors

Jerry A. Carter, Science Applications International Corporation
Michael Pickering, Pacific-Sierra Research Corporation
Henry Swanger, Science Applications International Corporation

Trademarks

IBM is a registered trademark of International Business Machines Corporation.
ORACLE is a registered trademark of Oracle Corporation.
SQL*Plus is a registered trademark of Oracle Corporation.
UNIX is a registered trademark of UNIX System Labs, Inc.

Ordering Information

This document was issued by the Monitoring Systems Operation of Science Applications International Corporation (SAIC) as part of the International Data Centre (IDC) Documentation. The ordering number for this document is SAIC-99/3022, (PSR-99/TN1145), published March 1999. Copies of this document may be ordered by FAX: (619) 458-4993.

This document is cited within other IDC documents as [IDC5.1.2].

Database Tutorial

CONTENTS

About this Document	i
■ PURPOSE	ii
■ SCOPE	ii
■ AUDIENCE	ii
■ RELATED INFORMATION	ii
■ USING THIS DOCUMENT	iv
Conventions	v
Introduction	1
■ STRUCTURED QUERY LANGUAGE	2
■ RELATIONAL DATABASES	2
■ TUTORIAL DATABASE	5
DBA Instructions	5
SQL Commands	7
■ CONNECTING TO DATABASE ACCOUNTS	8
Starting SQL*Plus	8
Changing Accounts	8
Terminating Connections	9
■ SELECTING DATA FROM TABLES	9
■ ORDERING ROWS OF RESULTS	18
■ ELIMINATING DUPLICATE ROWS	21
■ QUERYING MULTIPLE TABLES – JOINS	25
Cartesian Products	30
■ SUBQUERIES	31
■ OUTER JOINS	35

■ CREATING TABLES	39
■ CHANGING TABLE CONTENTS	41
INSERT	41
DELETE	43
UPDATE	43
ROLLBACK and COMMIT	44
SQL*Plus Extensions	47
■ QUERY BUFFER	48
■ CHARACTER FUNCTIONS	55
■ NUMBER FUNCTIONS	57
■ MANIPULATING DATES AND TIMES	60
Selecting Dates	62
Converting between Epoch Times and Dates	63
Improving Query Performance	67
■ USING INDEXED COLUMNS	69
■ LISTING MOST RESTRICTIVE TABLES LAST	73
■ USING IN VERSUS EXISTS	74
Navigating Databases	79
■ INSTANCES	80
■ ACCOUNTS	80
■ TABLES	81
Advanced Seismology Queries	85
■ FINDING ALL EVENTS IN ARRIVAL TIME WINDOWS	86
■ RETRIEVING ORIGIN INFORMATION FROM EVENTS WITH DEPTH PHASES	89
■ ESTIMATING STATION RESIDUALS	89
■ CALCULATING AZIMUTH RESOLUTION	90
■ PERFORMING LINEAR REGRESSION	92
Advanced Radionuclide Queries	97
■ SEARCHING FOR UNREVIEWED FULL SPECTRA	99

■ VERIFYING RECEIPT OF SPECTRA	100
■ SEARCHING FOR SPECIFIC NUCLIDES	101
■ DETERMINING CONCENTRATION RANGES	102
■ SEARCHING FOR SPECIFIC PEAKS	103

References	105
-------------------	-----

Glossary	G1
-----------------	----

Index	I1
--------------	----

Database Tutorial

TABLES

TABLE I:	TYPOGRAPHICAL CONVENTIONS	v
TABLE II:	TERMINOLOGY	vi
TABLE 1:	ORIGIN	2
TABLE 2:	COMPARISON OPERATORS	18
TABLE 3:	ORACLE GROUP-VALUE FUNCTIONS	22
TABLE 4:	SQL*PLUS QUERY BUFFER COMMANDS	54
TABLE 5:	CHARACTER FUNCTIONS	55
TABLE 6:	ORACLE SINGLE-VALUE FUNCTIONS	57
TABLE 7:	ORACLE LIST FUNCTIONS	58
TABLE 8:	IDC FUNCTIONS	58
TABLE 9:	ORACLE DATE AND TIME FUNCTIONS	60
TABLE 10:	ORACLE DATE AND TIME FORMATS	60

About this Document

This section describes the organization and content of the document and includes the following topics:

- Purpose
- Scope
- Audience
- Related Information
- Using this Document

About this Document

PURPOSE

This document describes how to use the Structured Query Language (SQL) to access and extract information from the database tables at the International Data Centre (IDC).

SCOPE

This document introduces SQL*Plus commands that are used for obtaining and manipulating data within the databases of the IDC. It also introduces the anatomy of a database table and explains where to obtain information on the contents and relationships of the database tables. It does not describe the database schema or the organization of the databases.

AUDIENCE

This document is intended for scientists, technicians, and managers who operate, maintain, and/or use the IDC and the data products that it provides.

RELATED INFORMATION

Use this document in conjunction with [IDC5.1.1Rev1], *Database Schema*.

See "References" on page 105 for a listing of all the sources of information consulted in preparing this document. Among those references are the documents described below.

[ANS86] outlines the syntax for the SQL Language standard.

E. F. Codd [Cod90] formally proposed the relational model in 1969. Between 1968 and 1978 he published many papers on the relational model. From 1979–1988 he proposed extensions to the original relational model (hence *Version 2* in the title). This book consolidates the writings of two decades into a single reference and is intended for Database Administrators (DBAs) and developers.

Chris Date was on the IBM technical team that designed two of the first commercially marketed relational products: SQL/DS and DB2. Since that time he has been a database consultant. [Dat86] is a collection of articles he has written. It is intended for technical readers.

[Eme89] contains an introduction to relational databases, relational database design, and SQL. It is intended for novice SQL users and developers.

[Fle89] provides a practical approach and methodologies for designing tables. This handbook is a standard on how to make relational theory work in practice.

[Gil89] focuses on the performance differences between correlated (EXISTS) and uncorrelated (IN) subqueries. [Sch88] includes guidelines for ordering items in SQL clauses, such as the order of columns in the FROM clause and the order of predicates in the WHERE clause.

[Gru90], [Hur88], and [Lus88] are three of the better introductions to the SQL language. The first two books are simpler; the latter is intended for the experienced programmer or manager. Also of note is [van88a], which contains an introduction to SQL formulated around the creation of a sports club database. This introductory guide is geared for the novice and focuses on American National Standards Institute (ANSI) SQL standard queries. [van88b] is a companion guide to [van88a] and is a more readable version of the SQL standard than [ANS86].

The IDC uses an ORACLE database management system as well as the technical publications written by Oracle Corporation. [Koc97] provides a comprehensive reference for ORACLE releases 7 through 8 and includes examples of SQLPLUS, query optimization, programming interfaces, and more.

This document is based on [And90b]. The text herein is an update of that document.

USING THIS DOCUMENT

This document is organized as follows:

- Introduction
This chapter introduces SQL and relational databases.
- SQL Commands
This chapter describes the basic SQL commands for obtaining and manipulating the information contained in the databases.
- SQL*Plus Extensions
This chapter introduces the ORACLE extensions of SQL (SQL*Plus).
- Improving Query Performance
This chapter suggests methods for improving query performance.
- Navigating Databases
This chapter describes how the tables of the IDC databases are distributed among computers and accounts and where the information on accounts and tables is located.
- Advanced Seismology Queries
This chapter contains examples of advanced queries used when manipulating seismological data.
- Advanced Radionuclide Queries
This chapter contains examples of advanced queries used when manipulating radionuclide data.
- References
This section lists the sources cited in this document.
- Glossary
This section defines the terms, abbreviations, and acronyms used in this document.

■ Index

This section lists topics and features provided in this document along with page numbers for reference.

Conventions

This document uses a variety of conventions, which are described in the following tables. Table I shows the typographical conventions. Table II explains certain technical terms that are not part of the standard Glossary, which is located at the end of this document. Most terms in this table pertain to SQL.

TABLE I: TYPOGRAPHICAL CONVENTIONS

Element	Font	Example
database table	bold	dataready
database table and attribute when written in the dot notation		prodtrack.status
headings, figure titles, and table titles		About this Document
the first time a SQL command is used in an example		select
attributes of database tables when written separately	<i>italics</i>	<i>status</i>
processes and software units		<i>ParseSubs</i>
user-defined arguments		<i>delete-remarks object</i>
computer code and output	courier	>(list 'a 'b 'c)
filenames, directories, and websites		amp.par
text that should be typed in exactly as shown		select
ORACLE key words in the text of the document	CAPS	SELECT
All rows of the database are not shown, due to space limitations.		<more rows at the bottom>

TABLE II: TERMINOLOGY

Term	Description
account	group of unique tables that are the default tables for queries
Cartesian product	query result that returns every possible combination of all rows in all tables in the FROM clause (most likely an error)
correlation name	alias for the table name that fully qualifies the field names in SQL clauses (providing an easy way of specifying exactly which fields come from which tables)
database instance	unique set of database accounts
join	query that allows the selection of data from more than one table and combines the data returned into a single result table
key	column or set of columns that make each row of a database table unique
natural join	query that returns rows from a join having the specified join field value in both tables
outer join	query that returns rows from a join having the join field value in one table but not the other
predicates	search conditions that are part of the SQL WHERE clause (contains a comparison operator that narrows the search to specific rows)
sequence numbers	number in an ORDER BY clause that refers to the order of the columns listed in the SELECT clause
subquery	complete SQL statement on the right-hand side of a search predicate in the WHERE clause

TABLE II: TERMINOLOGY (CONTINUED)

Term	Description
transaction	operation made on a database table that changes the table in some manner
tuple	row or record in a table
view	pseudo-table derived from one or more tables, which can be queried in the same manner as a table

Introduction

This chapter reviews the principles of relational databases and introduces the databases used for this tutorial. It includes the following topics:

- Structured Query Language
- Relational Databases
- Tutorial Database

Introduction

STRUCTURED QUERY LANGUAGE

SQL is a language for manipulating data in a relational database. Originally created by IBM, many dialects of SQL have been developed by other database vendors. In the early 1980s, the ANSI started the development of a language standard for managing relational data. The SQL standard was published in 1986 [ANS86]. The International Standards Organization (ISO) has published a standard as well. SQL*Plus is ORACLE's interactive SQL dialect. SQL*Plus includes extensions that are not part of the SQL standard.

This tutorial demonstrates how to use SQL*Plus to access the IDC databases. It assumes no prior knowledge of relational databases or SQL*Plus. A succession of progressively more complex SQL queries is used to demonstrate each command. Complete information on each command is available in [Koc97]. Except where noted, all queries comply with the ANSI SQL standard.

RELATIONAL DATABASES

A relational database is composed of tables, also called relations. Table 1 is the **origin** table as defined in [IDC5.1.1Rev1].

TABLE 1: ORIGIN

Column	Storage Type	Description
1 <i>orid</i>	number(8)	origin identifier
2 <i>lat</i>	float(24)	estimated latitude
3 <i>lon</i>	float(24)	estimated longitude
4 <i>depth</i>	float(24)	estimated depth

TABLE 1: ORIGIN (CONTINUED)

Column	Storage Type	Description
5 <i>time</i>	float(53)	epoch time
6 <i>evid</i>	number(8)	event identifier
7 <i>jdate</i>	number(8)	Julian date
8 <i>nass</i>	number(4)	number of associated phases
9 <i>ndef</i>	number(4)	number of locating phases
10 <i>ndp</i>	number(4)	number of depth phases
11 <i>grn</i>	number(8)	geographic region number
12 <i>srn</i>	number(8)	seismic region number
13 <i>etype</i>	varchar2(7)	event type
14 <i>depdp</i>	float(24)	estimated depth from depth phases
15 <i>dtype</i>	varchar2(1)	depth method used
16 <i>mb</i>	float(24)	body wave magnitude
17 <i>mbid</i>	number(8)	body wave magnitude identifier
18 <i>ms</i>	float(24)	surface wave magnitude
19 <i>msid</i>	number(8)	surface wave magnitude identifier
20 <i>ml</i>	float(24)	local magnitude
21 <i>mlid</i>	number(8)	local wave magnitude identifier
22 <i>algorithm</i>	varchar2(15)	location algorithm used
23 <i>auth</i>	varchar2(15)	source/originator
24 <i>commid</i>	number(8)	comment identifier
25 <i>lddate</i>	date	load date

Introduction ▼

A table is made up of columns (vertical) and rows (horizontal). For example, the **origin** table has 25 columns. The number of rows depends on the size of the database. The first few rows of an **origin** table are listed below:

LAT	LON	DEPTH	TIME	ORID	EVID	JDATE	...
42.7671	145.3814	51.1940	633669305.144	1115	-1	1990030	...
-6.5238	131.6238	57.1479	633670953.664	1116	-1	1990030	...
-17.3021	-178.5890	58.8421	633674762.818	1118	-1	1990030	...
-13.8659	173.5236	100.7905	633678421.987	1119	-1	1990030	...
-9.2605	125.4961	145.9096	633690606.505	1123	-1	1990030	...

<more rows at the bottom>

Columns are sometimes called attributes or fields, and rows are sometimes called tuples or records.

Each row in a table is unique. Although the combination of all attributes in a record is unique, in most cases a single attribute or a combination of a few attributes is guaranteed to be unique. The attribute or combination of a few attributes that are unique and are commonly used to reference a single record are known as keys. In the **origin** table, the *orid* attribute is a key as is the collection of attributes: *lat*, *lon*, *depth*, and *time*. A table may have more than one key. One of the keys is usually designated the primary key, and the others are known as alternate keys. Primary and alternate keys may not be NULL (otherwise they would not be unique). A table will usually also contain foreign keys, which are primary or alternate keys in some other table in the database. Foreign keys need not be unique and may be NULL. In the **origin** table, *evid* and *commid* are foreign keys.

The power of a relational database is its ability to relate information in one table to information in another table; this is accomplished mostly through keys. When the tables are designed, the information that is to be stored in the database is distributed logically among several tables. In a seismological example, consider the problem of arrivals, origins, and events. Arrivals are recorded by stations of a seismic network. Information from several arrivals (for example, arrival time) is combined to form a hypothesis for the event location, which is known as the origin. Several location hypotheses (origins) may be made for every event. Four tables are used to represent this information: **arrival**, **assoc**, **origin**, and **event**. Each arrival may con-

tribute to zero or more origins, and each origin requires several arrivals. The **assoc** table links (or associates) the arrival information to the origin information through the *arid* and *orid* keys. For any particular *orid* in the **origin** table, you can list all of the entries in the **assoc** table containing that *orid*. The **assoc** records list the associated *arids*, which can be used to obtain the **arrival** records. Conversely, you could find all of the origins that a particular arrival contributed by listing all of the **assoc** entries with a specific *arid*. The **assoc** records list the associated *orids*, which can be used to obtain the **origin** records. The **origin** records also contain the *evid* foreign key. Several **origin** records may have the same *evid*. Together these **origin** records contain the hypothesized locations for the particular event. The **event** table itself contains an *orid*, which identifies the preferred event solution.

TUTORIAL DATABASE

The database account used for the majority of the queries in this document can be installed and used to practice the queries introduced. The account is *geodemo* and it contains tables with data from the Prototype International Data Centre (PIDC). Not all of the queries will produce the same results, however. Those queries that use larger data sets, such as those in the chapters on advanced seismological and radionuclide queries, were run on larger data sets or on tables that are not included in the *geodemo* account. The *geodemo* data are not required to make this tutorial useful. The same queries may be run with any IDC data set.

A valid ORACLE account and password are required to run the SQL queries in this tutorial. The DBA will provide a database instance, an ORACLE account, and detailed instructions on how to connect to the database. You will also be issued a database password.

DBA Instructions

Use the following UNIX command to load the *geodemo* tutorial data set:

```
geodemo account/password
```

Introduction ▼

The *geodemo* program uses the ORACLE import command to load data into the account. The import commands that appear on the screen may be ignored. After the *geodemo* data set has been loaded, the data stay in the database until they are unloaded with the following UNIX command:

```
geodemodrop account/password
```

SQL Commands

This chapter introduces standard SQL commands and includes the following topics:

- Connecting to Database Accounts
- Selecting Data from Tables
- Ordering Rows of Results
- Eliminating Duplicate Rows
- Computing Functions on Groups of Rows
- Querying Multiple Tables – Joins
- Subqueries
- Outer Joins
- Creating Tables
- Changing Table Contents

SQL Commands

CONNECTING TO DATABASE ACCOUNTS

SQL*Plus can be considered an interactive program because it must be manually started and terminated. After starting the program, you can change the account.

Starting SQL*Plus

Enter the following command at the UNIX prompt to start SQL*Plus:

```
sqlplus account@instance
```

A password prompt is displayed:

```
Enter password:
```

After you have entered the correct password (the password will not be echoed to the screen) an ORACLE database banner is displayed, followed by a new prompt:

```
SQL>
```

In this document, this prompt indicates that you may enter the next SQL query.

When starting SQL*Plus, a login startup file is executed. This file sets your user environment and defines commonly used commands. At the IDC, a global login startup file is provided for all users.

Changing Accounts

The CONNECT command changes the connection from one database account to another:

```
connect account@instance
```

As when the initial connection was made, a prompt for a password is displayed, and you must enter a valid password before the connection is allowed:

```
Enter password:
```

Terminating Connections

The EXIT command terminates the database connection:

```
SQL> exit
```

SELECTING DATA FROM TABLES

Selecting data from the database is the most common SQL operation. A SELECT command consists of two or more clauses terminated by a semicolon (;):

```
select  some columns  
from    a table;
```

The SELECT clause is always entered first, immediately followed by the FROM clause. SQL key words, tables, and fields can be entered in lowercase or uppercase. This document shows example tables and attributes in lowercase. Words can be separated by spaces or tabs and can be carried across multiple lines. This document displays each SQL clause on a separate line. A semicolon must always terminate the statement.

Query (1) selects a number of columns from the **origin** table. Your query output may differ from this output. For example, the format of decimal numbers may not be identical. Column formats are controlled by SQL*Plus commands (see “SQL*Plus Extensions” on page 47). An alternative way of displaying *time* is discussed in “Manipulating Dates and Times” on page 60.

SQL ▼
Commands

(1) SQL> **select** lat, lon, depth, time, orid, evid, jdate
from origin;

LAT	LON	DEPTH	TIME	ORID	EVID	JDATE
0.9815	131.5063	4.6399	636710596.450	3499	-1	1990065
36.8840	73.3430	19.7271	636714964.102	3679	-1	1990065
-6.4516	148.4892	0.0000	636715738.291	3680	-1	1990065
1.2863	122.3456	0.0000	636719076.120	3681	-1	1990065
-5.9693	147.6910	196.9389	636721535.597	3503	-1	1990065
58.2493	26.7973	0.0000	636721753.664	3504	-1	1990065
59.1539	27.1154	0.0000	636723173.244	3682	-1	1990065
12.1327	143.6159	22.1815	636723655.160	3506	-1	1990065
-8.4487	150.8799	0.0000	636725410.723	3507	-1	1990065
-19.3829	-177.0614	70.8751	636726236.684	3508	-1	1990065
-20.4274	-67.2272	68.9936	636728262.655	3509	-1	1990065
-9.4297	125.8151	33.0000	636729740.586	3510	-1	1990065
-10.8992	117.5039	29.6973	636730261.597	3683	-1	1990065
36.8908	73.4689	9.1589	636732572.863	3512	-1	1990065
-17.8557	168.0396	27.5160	636733569.784	3513	-1	1990065
32.9234	74.9347	12.9065	636734589.275	3514	-1	1990065
21.9599	142.8868	0.0000	636734894.155	3684	-1	1990065
-10.5516	119.7778	37.6402	636737108.355	3516	-1	1990065

18 rows selected.

SQL>

The asterisk (*) character represents all columns. Query (2) would return the same number of rows as Query (1) but would display all columns of the **origin** table.

(2) SQL> **select** *
from origin;

The result of any query is a table of columns and rows. The order of columns in the SELECT clause determines the display sequence. If all fields are selected with SELECT *, they are output in the sequence that they were created.

Query (1) showed how the SELECT clause restricts which columns are displayed. Query (3) shows how the WHERE clause restricts which rows are returned.

```
(3) SQL> select arid, orid, sta, phase, delta, seaz
        from   assoc
        where  orid=3508;
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67869	3508	YKA	P	95.330	237.14
67669	3508	ASAR	P	45.601	94.47
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47
66946	3508	MAT	P	70.075	135.08
67522	3508	ASPA	P	45.601	94.47
67026	3508	ARA0	PKP	127.988	27.34
66814	3508	EKA	PKP	143.755	350.24

8 rows selected.

SQL>

The WHERE clause contains search conditions, called predicates, which narrow the search to specific rows. The predicate contains a comparison operator that compares two expressions (see Table 2 for a list of the comparison operators.) In Query (3), the *orid* column is the left-hand expression, and the constant 3508 is the right-hand expression. These expressions are tested for equality with the = operator, and only those rows containing an *orid* of 3508 are returned. The test for inequality uses the != (not equal) operator. The = operator also tests strings for equality. Query (4) shows how the text of the string must be enclosed by single quotes.

```
(4) SQL> select arid, orid, sta, phase, delta, seaz
        from   assoc
        where  sta = 'MAT';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
66946	3508	MAT	P	70.075	135.08
66949	3683	MAT	P	51.289	21.35
66951	3683	MAT	LR	51.289	21.35

SQL>

SQL ▼
Commands

Although SQL key words can be entered in either uppercase or lowercase, string searches are case sensitive depending on how data are stored in the database. For example, the range defined for the *sta* attribute is any uppercase string up to six characters (see [IDC5.1.1Rev1]). Because *sta* is stored in uppercase, the following WHERE clause condition would return no rows:

```
where sta='mat'
```

The LIKE operator matches partial strings. The underscore character (`_`) matches any single character. Query (5) uses two-letter phase names beginning with 'S' to limit the results.

```
(5) SQL> select  arid, orid, sta, phase, delta, seaz
      from      assoc
      where     phase like 'S_';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67021	3682	ARA0	Sn	10.404	356.88
71464	3682	KAF	Sn	2.985	352.71

```
SQL>
```

The percent sign (%) matches zero, one, or more characters. Query (6) obtains the information for all s-type phases.

(6) SQL> select arid, orid, sta, phase, delta, seaz
from assoc
where phase like 'S%';

ARID	ORID	STA	PHASE	DELTA	SEAZ
68225	3503	ASPA	S	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67672	3508	ASAR	ScP	45.601	94.47
67531	3510	ASPA	S	16.199	330.16
67095	3512	GAR	S	3.270	129.18
67542	3516	ASPA	S	18.793	311.86
68086	3679	GAR	S	3.199	312.33
67021	3682	ARA0	Sn	10.404	356.88
71464	3682	KAF	Sn	2.985	352.71
67092	3683	GAR	S	66.455	321.53

10 rows selected.

SQL>

Three logical operators (AND, OR, and NOT) may be used to combine multiple predicates in a WHERE clause. AND specifies that both predicates must be satisfied for a row of data to be returned, as shown in Query (7).

(7) SQL> select arid, orid, sta, phase, delta, seaz
from assoc
where orid=3508
and sta='ASAR';

ARID	ORID	STA	PHASE	DELTA	SEAZ
67669	3508	ASAR	P	45.601	94.47
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47

SQL>

SQL ▼
Commands

OR specifies that the row should be returned if either predicate is satisfied, as shown in Query (8).

```
(8) SQL> select  arid, orid, sta, phase, delta, seaz
        from    assoc
        where    orid=3508
        or      sta='ASAR';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67490	3499	ASAR	P	24.757	354.27
67506	3503	ASAR	P	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67510	3506	ASAR	P	37.037	15.89
67664	3507	ASAR	P	22.262	49.67
67869	3508	YKA	P	95.330	237.14
67669	3508	ASAR	P	45.601	94.47
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47
66946	3508	MAT	P	70.075	135.08
67522	3508	ASPA	P	45.601	94.47
67026	3508	ARA0	PKP	127.988	27.34
66814	3508	EKA	PKP	143.755	350.24
67680	3510	ASAR	P	16.199	330.16
67691	3512	ASAR	P	83.081	315.51
67692	3513	ASAR	P	32.364	86.21
67694	3513	ASAR	pP	32.364	86.21
67695	3514	ASAR	P	79.738	313.03
67697	3516	ASAR	P	18.793	311.86
67492	3679	ASAR	P	83.157	126.55
67494	3680	ASAR	P	22.199	217.62
67686	3683	ASAR	P	20.168	131.41

22 rows selected.

SQL>

Other comparison operators allow tests for a range of values. Query (9) tests for an *orid* < (less than) 3508.

```
(9) SQL> select  arid, orid, sta, phase, delta, seaz
        from    assoc
        where    orid < 3508
        and      sta = 'ASAR';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67490	3499	ASAR	P	24.757	354.27
67506	3503	ASAR	P	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67510	3506	ASAR	P	37.037	15.89
67664	3507	ASAR	P	22.262	49.67

SQL>

Query (10) tests for an *orid* <= (less than or equal to) 3508.

```
(10) SQL> select  arid, orid, sta, phase, delta, seaz
        from    assoc
        where    orid <= 3508
        and      sta = 'ASAR';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67490	3499	ASAR	P	24.757	354.27
67506	3503	ASAR	P	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67510	3506	ASAR	P	37.037	15.89
67664	3507	ASAR	P	22.262	49.67
67669	3508	ASAR	P	45.601	94.47
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47

8 rows selected.

SQL>

SQL ▼
Commands

Query (11) adds a search for a value \geq (greater than or equal to) 3500 to Query (10).

```
(11)  SQL> select  arid, orid, sta, phase, delta, seaz
        from      assoc
        where      orid  $\geq$  3500
        and        orid  $\leq$  3508
        and        sta = 'ASAR';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67506	3503	ASAR	P	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67510	3506	ASAR	P	37.037	15.89
67664	3507	ASAR	P	22.262	49.67
67669	3508	ASAR	P	45.601	94.47
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47

7 rows selected.

SQL>

The BETWEEN operator used in Query (12) offers a shortcut for Query (11).

```
(12)  SQL> select  arid, orid, sta, phase, delta, seaz
        from      assoc
        where      orid between 3500 and 3508
        and        sta = 'ASAR';
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67506	3503	ASAR	P	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67510	3506	ASAR	P	37.037	15.89
67664	3507	ASAR	P	22.262	49.67
67669	3508	ASAR	P	45.601	94.47
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47

7 rows selected.

SQL>

In Query (13) the IN operator searches for a specific list of values.

```
(13) SQL> select  arid, orid, sta, phase, delta, seaz
        from    assoc
        where   orid between 3500 and 3508
        and     sta = 'ASAR'
        and     phase in ('P','S');
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67506	3503	ASAR	P	22.114	39.02
67507	3503	ASAR	S	22.114	39.02
67510	3506	ASAR	P	37.037	15.89
67664	3507	ASAR	P	22.262	49.67
67669	3508	ASAR	P	45.601	94.47

SQL>

You can specify the reverse of any operator, as shown in Query (14), where NOT IN specifies the reverse of IN.

```
(14) SQL> select  arid, orid, sta, phase, delta, seaz
        from    assoc
        where   orid between 3500 and 3508
        and     sta = 'ASAR'
        and     phase not in ('P','S');
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67671	3508	ASAR	PcP	45.601	94.47
67672	3508	ASAR	ScP	45.601	94.47

SQL>

Table 2 summarizes the comparison operators introduced in this section.

TABLE 2: COMPARISON OPERATORS

Operator	Sample 'WHERE' Clause
=	where orid = 3508
>	where orid > 3508
>=	where orid >= 3508
<	where orid < 3508
<=	where orid <= 3508
BETWEEN	where orid between 3508 and 3510
IN	where phase in ('Pn', 'Sn', 'Lg')
NOT IN	where orid not in ('Pn', 'Sn', 'Lg')
LIKE	where phase like 'P_'

ORDERING ROWS OF RESULTS

In the previous examples, the rows of query results were displayed in an order determined by ORACLE. You can specify the order by using the ORDER BY clause. Query (15) sorts the result by phase.

```
(15)  SQL> select  arid, orid, sta, phase, delta, seaz
        from      assoc
        where     orid=3508
        and       phase like 'P%'
        order by  phase;
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67869	3508	YKA	P	95.330	237.14
67669	3508	ASAR	P	45.601	94.47
66946	3508	MAT	P	70.075	135.08
67522	3508	ASPA	P	45.601	94.47
67026	3508	ARA0	PKP	127.988	27.34

```

        66814      3508 EKA      PKP      143.755  350.24
        67671      3508 ASAR     PcP      45.601   94.47
7 rows selected.

```

SQL>

Query (16) references more than one column in the ORDER BY clause:

```

(16)  SQL> select   arid, orid, sta, phase, delta, seaz
        from       assoc
        where      orid=3508
        and        phase like 'P%'
        order by   phase, delta;

```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67669	3508	ASAR	P	45.601	94.47
67522	3508	ASPA	P	45.601	94.47
66946	3508	MAT	P	70.075	135.08
67869	3508	YKA	P	95.330	237.14
67026	3508	ARA0	PKP	127.988	27.34
66814	3508	EKA	PKP	143.755	350.24
67671	3508	ASAR	PcP	45.601	94.47

7 rows selected.

SQL>

By default, ORACLE sorts in ascending order. You can specify that the results be sorted in descending order or in a combination of both ascending and descending order. Query (17) sorts first by *phase* in ascending (ASC) order, then by *delta* in descending (DESC) order.

SQL ▼
Commands

```
(17)  SQL> select    arid, orid, sta, phase, delta, seaz
        from      assoc
        where     orid=3508
        and       phase like 'P%'
        order by  phase asc, delta desc;
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67869	3508	YKA	P	95.330	237.14
66946	3508	MAT	P	70.075	135.08
67669	3508	ASAR	P	45.601	94.47
67522	3508	ASPA	P	45.601	94.47
66814	3508	EKA	PKP	143.755	350.24
67026	3508	ARA0	PKP	127.988	27.34
67671	3508	ASAR	PcP	45.601	94.47

7 rows selected.

SQL>

You can replace columns in the ORDER BY clause with sequence numbers that correspond to the position of the column in the SELECT clause. For example, in Query (17) *phase* is the fourth column, and *arid* is the first column. Query (18) produces the same result by referencing column numbers.

```
(18)  SQL> select    arid, orid, sta, phase, delta, seaz
        from      assoc
        where     orid=3508
        and       phase like 'P%'
        order by  4 asc, 5 desc;
```

ARID	ORID	STA	PHASE	DELTA	SEAZ
67869	3508	YKA	P	95.330	237.14
66946	3508	MAT	P	70.075	135.08
67669	3508	ASAR	P	45.601	94.47
67522	3508	ASPA	P	45.601	94.47
66814	3508	EKA	PKP	143.755	350.24
67026	3508	ARA0	PKP	127.988	27.34
67671	3508	ASAR	PcP	45.601	94.47

7 rows selected.

SQL>

ELIMINATING DUPLICATE ROWS

Sometimes a query will return duplicate rows. Query (19) selects all *orids* less than 3505 from the **assoc** table and consequently returns many duplicate rows.

```
(19)  SQL> select  orid
        from    assoc
        where   orid < 3505;
```

```
      ORID
-----
      3499
      3499
      3503
      3503
      3503
      3503
      3503
      3503
      3503
      3504
      3504
      3504
      3504
```

12 rows selected.

SQL>

Specifying **DISTINCT** in the **SELECT** clause eliminates duplicate rows, as shown in Query (20).

```
(20)  SQL> select  distinct orid
        from    assoc
        where   orid < 3505;
```

```
      ORID
-----
      3499
      3503
      3504
```

SQL>

Computing Functions on Groups of Rows

Group-value functions compute summary information across groups of rows. Table 3 lists common group functions, which appear in the SELECT clause and usually take column names as arguments.

TABLE 3: ORACLE GROUP-VALUE FUNCTIONS

Function	Definition
AVG(<i>value</i>)	average of <i>value</i> for a group of rows
COUNT(<i>value</i>)	count of the number of rows in a group of rows
MAX(<i>value</i>)	maximum of <i>value</i> for a group of rows
MIN(<i>value</i>)	minimum of <i>value</i> for a group of rows
STDDEV(<i>value</i>)	standard deviation of <i>value</i> for a group of rows
SUM(<i>value</i>)	sum of <i>value</i> for a group of rows
VARIANCE(<i>value</i>)	variance of <i>value</i> for a group of rows

The GROUP BY clause limits the rows to which the function is applied. The group-value function is applied to each group of rows for which the value of the GROUP BY column is unique.

Query (21) counts the number of events by day.

```
(21) SQL> select    jdate, count(oid)
        from      origin
        group by   jdate;

      JDATE COUNT(ORID)
-----
1990065      18

SQL>
```

Query (22) counts the number of *arids* for each *orid* in the **assoc** table.

```
(22)  SQL> select  orid, count(arid)
        from      assoc
        group by  orid;
```

ORID	COUNT(ARID)
3499	2
3503	6
3504	4
3506	12
3507	2
3508	8
3509	3
3510	4
3512	5
3513	8
3514	4
3516	7
3679	10
3680	2
3681	4
3682	10
3683	25
3684	4

18 rows selected.

SQL>

Adding a WHERE clause reduces the number of source rows processed for the aggregate count as shown in Query (23).

SQL ▼
Commands

```
(23)  SQL> select   orid, count(arid)
        from      assoc
        where     orid < 3510
        group by  orid;
```

ORID	COUNT(ARID)
3499	2
3503	6
3504	4
3506	12
3507	2
3508	8
3509	3

7 rows selected.

SQL>

The HAVING clause restricts the GROUP BY clause in the same way that the WHERE clause restricts the SELECT clause. Query (24) returns only those *orids* that have more than two *arids*.

```
(24)  SQL> select   orid, count(arid)
        from      assoc
        where     orid < 3510
        group by  orid
        having    count(arid) > 2;
```

ORID	COUNT(ARID)
3503	6
3504	4
3506	12
3508	8
3509	3

SQL>

Query (25) sorts the final result in decreasing order by the *orid* with the most *arids*. It uses all of the SQL SELECT clauses: SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY. In this query, the ORDER BY clause instructs ORACLE to sort by the second column, *count(arid)*, in the SELECT clause.

```
(25)  SQL> select   orid, count(arid)
        from      assoc
        where     orid < 3510
        group by  orid
        having    count(arid) > 2
        order by  2 desc;
```

ORID	COUNT(ARID)
3506	12
3508	8
3503	6
3504	4
3509	3

```
SQL>
```

QUERYING MULTIPLE TABLES – JOINS

In previous examples, single tables were queried; however, the data will not always be available in one table. A “join” selects data from more than one table and returns a single table as a result. This section describes two types of joins: a natural join and an outer join. A natural join, which is the most common join, returns rows that have the join field value in all tables. An outer join returns rows that have the join field value in one table but not in the others (see “Outer Joins” on page 35). Query (26) is a single-table (**origin**) query that finds all origins in a given latitude, longitude window; however, the table does not contain *phase*. To find all phases associated with the origins, the **assoc** table, which contains *phase*, is needed. Both tables contain the *orid* column, which provides a link to join them. Query (27) uses the dot notation to specify this link in the WHERE clause. In dot notation, a dot separates the table name from the column name (**table.column**).

SQL ▼
Commands

```
(26)  SQL> select  lat, lon, depth, time
        from      origin
        where     lat between 35.0 and 40.0
        and       lon between 50.0 and 75.0;
```

LAT	LON	DEPTH	TIME
36.8840	73.3430	19.7271	636714964.102
36.8908	73.4689	9.1589	636732572.863

SQL>

```
(27)  SQL> select  arid, lat, lon, depth, time, phase
        from      assoc, origin
        where     assoc.orid=origin.orid
        and       lat between 35.0 and 40.0
        and       lon between 50.0 and 75.0;
```

ARID	LAT	LON	DEPTH	TIME	PHASE
67600	36.8908	73.4689	9.1589	636732572.863	P
67093	36.8908	73.4689	9.1589	636732572.863	P
67095	36.8908	73.4689	9.1589	636732572.863	S
67891	36.8908	73.4689	9.1589	636732572.863	P
67691	36.8908	73.4689	9.1589	636732572.863	P
68114	36.8840	73.3430	19.7271	636714964.102	P
68117	36.8840	73.3430	19.7271	636714964.102	pP
67370	36.8840	73.3430	19.7271	636714964.102	P
66866	36.8840	73.3430	19.7271	636714964.102	P
67439	36.8840	73.3430	19.7271	636714964.102	P
67492	36.8840	73.3430	19.7271	636714964.102	P
67011	36.8840	73.3430	19.7271	636714964.102	P
68083	36.8840	73.3430	19.7271	636714964.102	P
68086	36.8840	73.3430	19.7271	636714964.102	S
67834	36.8840	73.3430	19.7271	636714964.102	P

15 rows selected.

SQL>

When a column in the SELECT clause occurs in more than one table in the FROM clause, you must specify from which table the column should be displayed. Query (28) adds *orid* to Query (27)'s SELECT clause and specifies that the *orid* from the original table should be displayed.

```
(28) SQL> select  origin.orid, arid, lat, lon, depth, time, phase
        from    assoc, origin
        where   assoc.orid=origin.orid
        and     lat between 35.0 and 40.0
        and     lon between 50.0 and 75.0;
```

ORID	ARID	LAT	LON	DEPTH	TIME	PHASE
3512	67600	36.8908	73.4689	9.1589	636732572.863	P
3512	67093	36.8908	73.4689	9.1589	636732572.863	P
3512	67095	36.8908	73.4689	9.1589	636732572.863	S
3512	67891	36.8908	73.4689	9.1589	636732572.863	P
3512	67691	36.8908	73.4689	9.1589	636732572.863	P
3679	68114	36.8840	73.3430	19.7271	636714964.102	P
3679	68117	36.8840	73.3430	19.7271	636714964.102	pP
3679	67370	36.8840	73.3430	19.7271	636714964.102	P
3679	66866	36.8840	73.3430	19.7271	636714964.102	P
3679	67439	36.8840	73.3430	19.7271	636714964.102	P
3679	67492	36.8840	73.3430	19.7271	636714964.102	P
3679	67011	36.8840	73.3430	19.7271	636714964.102	P
3679	68083	36.8840	73.3430	19.7271	636714964.102	P
3679	68086	36.8840	73.3430	19.7271	636714964.102	S
3679	67834	36.8840	73.3430	19.7271	636714964.102	P

15 rows selected.

SQL>

Query (28) used the name of the **origin** table to fully qualify *orid*. A field may also be qualified by using a correlation name, which is like an alias for the actual table. Query (29) uses correlation names to specify which fields come from which tables. This query would return the same result as Query (28).

SQL ▼ Commands

```
(29)  SQL> select  o.orid, a.arid, o.lat, o.lon, o.depth, o.time, a.phase
        from      assoc a, origin o
        where     a.orid=o.orid
        and       o.lat between 35.0 and 40.0
        and       o.lon between 50.0 and 75.0;
```

The correlation name for the **assoc** table is **a** and the correlation name for the **origin** table is **o**. Query (29) specifies which attributes are to be selected from which tables. This feature is useful especially in complex queries.

Correlation names may consist of multiple characters but may not conflict with an SQL key word. For example, **ar** is a valid correlation name for the **arrival** table, but **as** is not a valid name for the **assoc** table. Query (30) results in an error, because the correlation name **as** conflicts with an SQL key word.

```
(30)  SQL> select  o.orid, as.arid, o.lat, o.lon, o.time, as.phase,
        ar.time, ar.azimuth, ar.slow
        from      assoc as, arrival ar, origin o
        where     as.orid=o.orid
        and       as.arid=ar.arid
        and       o.lat between 35.0 and 40.0
        and       o.lon between 50.0 and 75.0;
```

```
select  o.orid, as.arid, o.lat, o.lon, o.time, as.phase,
        *
ERROR at line 1:
ORA-00936: missing expression

SQL>
```

Likewise, **or** is an invalid correlation name for **origin**. Any number of tables may be joined. Query (31) builds on Query (29) and includes information from the **arrival** table.

```
(31) SQL> select  o.orid, a.arid, o.lat, o.lon, o.time, a.phase, ar.time,
                ar.azimuth, ar.slow
                from    assoc a, arrival ar, origin o
                where    a.orid=o.orid
                and      a.arid=ar.arid
                and      o.lat between 35.0 and 40.0
                and      o.lon between 50.0 and 75.0;
```

ORID	ARID	LAT	LON	TIME	PHASE	TIME	AZIMUTH	SLOW
3679	66866	36.8840	73.3430	636714964.102	P	636715402.398	78	3.15
3679	67011	36.8840	73.3430	636714964.102	P	636715430.148	96	7.26
3512	67093	36.8908	73.4689	636732572.863	P	636732624.094	-1	-1.00
3512	67095	36.8908	73.4689	636732572.863	S	636732665.000	-1	-1.00
3679	67370	36.8840	73.3430	636714964.102	P	636715455.398	97	6.86
3679	67439	36.8840	73.3430	636714964.102	P	636715695.703	326	5.01
3679	67492	36.8840	73.3430	636714964.102	P	636715708.000	326	5.01
3512	67600	36.8908	73.4689	636732572.863	P	636733305.906	326	5.01
3512	67691	36.8908	73.4689	636732572.863	P	636733317.797	315	5.90
3679	67834	36.8840	73.3430	636714964.102	P	636715693.797	351	5.41
3512	67891	36.8908	73.4689	636732572.863	P	636733304.203	350	5.44
3679	68083	36.8840	73.3430	636714964.102	P	636715013.000	-1	-1.00
3679	68086	36.8840	73.3430	636714964.102	S	636715052.703	-1	-1.00
3679	68114	36.8840	73.3430	636714964.102	P	636715449.923	91	8.76
3679	68117	36.8840	73.3430	636714964.102	pP	636715454.326	-1	-1.00

15 rows selected.

SQL>

To summarize the join query:

- SELECT specifies which fields to display. If a field is in more than one table, specify which table to use. Query (28) selected an *orid* from the **origin** table. Query (29) used a correlation name to qualify each column.
- FROM lists all tables.
- A join predicate in the WHERE clause specifies the join column (the field appearing in more than one table). Query (27) showed a join predicate that joined **assoc** and **origin** on *orid*. Query (31) showed two join predicates: one joined **assoc** and **origin** on *orid*, and one joined **arrival** and **assoc** on *arid*.

Cartesian Products

All tables referenced in the FROM clause should have a join predicate in the WHERE clause. Otherwise, the result is a cartesian product consisting of every possible combination of all the rows in all the tables in the FROM clause. For example, in Query (32) the **origin** and **origerr** tables each have 18 rows. With a join on the *orid* column in the WHERE clause, a join returns a row count of 18. Without the join predicate, a cartesian product results in a row count of 324 (18 *origin* rows multiplied by 18 *origerr* rows).

```
(32)  SQL> select  count(*)
        from      origin, origerr
        where      origin.orid=origerr.orid;
```

```
      COUNT(*)
-----
          18
```

```
SQL> select  count(*)
        from      origin, origerr;
```

```
      COUNT(*)
-----
          324
```

```
SQL>
```

Query (33) and Query (34) were edited, but a table that was no longer needed was not removed from the FROM clause. Both queries ran on a data set containing eight weeks of data (the **arrival** table contains 46,856 rows, and the **assoc** table contains 4,396 rows). The ORACLE timing feature was used to gather performance statistics. Query (33) references the **arrival** table in the FROM clause, but the WHERE clause is missing a predicate joining it to **assoc**. It ran for more than 56 hours. By removing the reference to **arrival**, Query (34) ran for only 4 seconds.

```
(33)  SQL> set timing on;
      SQL> select    ac.phase, count(ac.arid)
      from          arrival ar, assoc ac
      where         ac.phase in ('Pn', 'Pg', 'Sn', 'Lg')
      group by      ac.phase;
```

```
PHASE      COUNT(AC.ARID)
-----
Lg          80311184
Pg          21085200
Pn          72533088
Sn          18320696
Elapsed: 56:25:20.45
```

```
SQL>
```

```
(34)  SQL> select    ac.phase, count(ac.arid)
      from          assoc ac
      where         ac.phase in ('Pn', 'Pg', 'Sn', 'Lg')
      group by      ac.phase;
```

```
PHASE      COUNT(AC.ARID)
-----
Lg          1714
Pg          450
Pn          1548
Sn          391
Elapsed: 00:00:04.07
```

```
SQL>
```

SUBQUERIES

A predicate contains two expressions: one to the left of the comparison operator and one to the right. The previous examples used either a constant or the name of a join column in the right-hand expression. This expression may also contain a complete SQL statement enclosed by parentheses. The value resulting from the nested SQL statement is then applied to the left-hand expression. Query (35) uses an IN subquery to find the earliest arrival time associated with origin 3679.

SQL ▼
Commands

```
(35)  SQL> select  min(time)
        from    arrival
        where   arid in
        (select  arid
        from    assoc
        where   orid=3679);
```

```
      MIN(TIME)
-----
636715013

SQL>
```

Query (36) shows the same query rewritten as an EXISTS subquery:

```
(36)  SQL> select  min(time)
        from    arrival
        where   exists
        (select  arid
        from    assoc
        where   arrival.arid=assoc.arid
        and     orid=3679);
```

```
      MIN(TIME)
-----
636715013

SQL>
```

Query (37) writes the same query as a straight join, demonstrating that a subquery often provides alternative syntax for a join.

```
(37)  SQL> select  min(time)
        from    arrival, assoc
        where   assoc.arid=arrival.arid
        and     assoc.orid=3679;
```

```
      MIN(TIME)
-----
636715013

SQL>
```

Subqueries may be nested within other subqueries to an unlimited number of levels. Query (38) counts all associated arrivals by *phase* for a given time period, including only those stations that are in the GSETT network.

```
(38)  SQL> select  sta, count(arid)
        from      assoc
        where      arid in
                (select arid
                 from   arrival
                 where  time between 636725500 and 636730000
                 and    sta in
                        (select sta
                         from   affiliation
                         where  net='GSETT'
                        )
                )
        group by   sta
        order by    sta;
```

STA	COUNT(ARID)
-----	-----
ARA0	2
ASAR	5
ASPA	2
EKA	1
MAT	1
WRA	2
YKA	2

7 rows selected.

```
SQL>
```

A subquery can select the difference between two sets to find elements in one that are not in the other. A join cannot provide this function. For example, to identify the number of unassociated arrivals, Query (39) establishes that the **arrival** table contains 368 rows.

SQL ▼
Commands

(39) SQL> select count(*) from arrival;

```

COUNT(*)
-----
        368

SQL>
```

Query (40) and Query (41) show how both a join and a subquery can count the number of associated arrivals, the first with a join and the second with an EXISTS subquery.

(40) SQL> select count(ar.arid)
 from arrival ar, assoc ac
 where ar.arid=ac.arid;

```

COUNT(AR.ARID)
-----
          120

SQL>
```

(41) SQL> select count(arid)
 from arrival
 where exists
 (select arid
 from assoc
 where arrival.arid=assoc.arid);

```

COUNT(ARID)
-----
          120

SQL>
```

A join, however, cannot find the number of arrivals that are not associated. The **arrival** table contains 368 rows, so Query (42)'s result is nonsense.

```
(42)  SQL> select  count(ar.arid)
        from      arrival ar, assoc ac
        where      ar.arid != ac.arid;
```

```
COUNT(AR.ARID)
-----
          44040
```

```
SQL>
```

Query (43) uses a NOT EXISTS subquery to obtain the correct answer.

```
(43)  SQL> select  count(arid)
        from      arrival
        where      not exists
                    (select  arid
                     from      assoc
                     where      arrival.arid=assoc.arid);
```

```
COUNT(ARID)
-----
          248
```

```
SQL>
```

The ability to identify the empty set, as shown in Query (43), is necessary for solving the problem of the outer join, the subject of the next section.

OUTER JOINS

An outer join is a join between two tables, which returns (a) all the rows matching a join condition plus (b) all the rows from one table that do not match the join condition. For those rows in (b), the queried columns from the other table are set to NULL.

To fully understand joins, it is important to understand the relationships between tables in the database. In a **one-to-many** relationship, each row in one table relates to many rows in another table. For example, every **origin** row has many related

SQL ▼
Commands

assoc rows. This relationship makes seismological sense because more than one phase is required to locate an event. Query (44) and Query (45) return different row counts for the same *orid*.

```
(44)  SQL> select  count(*)
        from      origin
        where     orid=3679;
```

```
COUNT(*)
-----
        1
```

```
SQL>
```

```
(45)  SQL> select  count(*)
        from      assoc
        where     orid=3679;
```

```
COUNT(*)
-----
       10
```

```
SQL>
```

Each row in the **assoc** table has one corresponding row (one-to-one relationship) in the **arrival** table.¹ However, some **arrival** phases are unassociated and do not have a corresponding **assoc** row. So, the relationship between **arrival** and **assoc** is one-to-one-or-none: every **arrival** row will have one or no **assoc** row. Query (39), Query (40), and Query (43) established that out of 368 arrivals, 120 arrivals were associated, and 248 arrivals were unassociated. As shown in Query (46), although **arrival** has 368 rows, a join between **arrival** and **assoc** returns only 120 rows, excluding all the unassociated arrivals.

1. This assumes that each **arrival** can have only one solution. This will not be the case for data sets that store intermediate results. In that case, every **arrival** row could have many **assoc** rows.

```
(46) SQL> select  ar.arid, a.phase, ar.time, ar.azimuth, ar.slow
           from    arrival ar, assoc a
           where   ar.arid=a.arid
           order by 1;
```

ARID	PHASE	TIME	AZIMUTH	SLOW
66814	PKP	636727398.297	-1	-1.00
66826	P	636730856.000	-1	-1.00

<many more rows>

120 rows selected.

SQL>

The natural join returns rows that have the same *arid* in both tables. The outer join returns these rows plus the rows having an *arid* in one table but not the other. For example, many **arrival** rows do not have a corresponding **assoc** row, because they are not associated with an event. Unassociated arrivals belong to the outer join of **arrival** and **assoc**. Query (47) uses the NOT EXISTS operator introduced in Query (43) to find rows in **arrival** that do not have a match in **assoc**.

```
(47) SQL> select  arid, time, azimuth, slow
           from    arrival
           where   not exists
                   (select  arid
                    from    assoc
                    where   arrival.arid=assoc.arid);
```

ARID	TIME	AZIMUTH	SLOW
67127	636710454.898	-1	-1.00
67007	636711053.703	31	5.17
67144	636711856.500	-1	-1.00
67146	636711883.102	-1	-1.00
66858	636713095.500	116	16.10
66859	636713130.398	110	25.84

<many more rows>

248 rows selected.

SQL>

SQL ▼
Commands

Query (46) and Query (47) provide information about all arrivals whether associated or unassociated. The two queries may be combined with the UNION operator to return all data in a single result table. The SELECT lists in the two UNIONed queries must have the same number of columns. Fields having no data, such as the *phase* field, which is in **assoc** but not **arrival**, need a place holder. Query (48) prints dashes for the *phase* field of the outer join rows.

```
(48) SQL> select    ar.arid, a.phase, ar.time, ar.azimuth, ar.slow
        from      arrival ar, assoc a
        where     ar.arid=a.arid
        union
        select    ar.arid, '-----', ar.time, ar.azimuth, ar.slow
        from      arrival ar
        where     not exists
                (select  arid
                 from    assoc
                 where   ar.arid=arid)
        order by  1;
```

ARID	PHASE	TIME	AZIMUTH	SLOW
66814	PKP	636727398.297	-1	-1.00
66826	P	636730856.000	-1	-1.00
66829	-----	636730479.906	-1	-1.00
66858	-----	636713095.500	116	16.10
66859	-----	636713130.398	110	25.84
66860	-----	636713815.297	279	15.87

<many more rows>
368 rows selected.

SQL>

The ORACLE outer join operator, +, is easier to use, but is not ANSI SQL. In Query (49) the outer join operator tags which table will not have data. For example, **assoc** will not have rows for some *arids* in **arrival**, so **assoc** is tagged in the join predicate.

```
(49)  SQL> select  ar.arid, a.phase, ar.time, ar.azimuth, ar.slow
        from      arrival ar, assoc a
        where     ar.arid=a.arid(+)
        order by  1;
```

ARID	PHASE	TIME	AZIMUTH	SLOW
66814	PKP	636727398.297	-1	-1.00
66826	P	636730856.000	-1	-1.00
66829		636730479.906	-1	-1.00
66858		636713095.500	116	16.10
66859		636713130.398	110	25.84
66860		636713815.297	279	15.87
66861		636713832.000	285	34.72
66862		636714355.797	149	15.87
66863		636714387.102	167	27.78
66864		636714565.398	168	15.22
66865		636714594.500	172	27.78
66866	P	636715402.398	78	3.15

<many more rows>

368 rows selected.

SQL>

CREATING TABLES

Up to this point, this document has focused on extracting data from existing tables. An often useful method is to create tables and populate them with intermediate results or with subsets of very large tables so that subsequent queries are more manageable. The *geodemo* account contains tables that were created as subsets of the operational database at the PIDC.

The CREATE TABLE command creates a new table. This command must include the name of the new table as well as definitions of the columns that will be included in the new table.

SQL ▼
Commands

One method of defining a new table is to provide an explicit definition:

```
SQL> create table name (  
      column1 datatype1 [constraint1]  
      column2 datatype2 [constraint2]  
      ...  
      );
```

Column1, *column2*, and so on are the names of the columns of the tables. *Datatypes* for the columns typically used at the IDC include VARCHAR2(*precision*) for characters, NUMBER(*precision*) for integers, FLOAT(*precision*) for floating point numbers, and DATE for ORACLE dates. *Precision* defines the number of digits that the number may have. *Constraints* such as NOT NULL and PRIMARY KEY can be placed on columns of a table. [Koc97] provides more information about these options.

Another method of creating a new table is to use the AS delimiter to generate the table from the results of a query:

```
SQL> create table name as select...  
      from...;
```

The column names and definitions for the new table are derived from the SELECT statement, and the table is populated with the results of the query. If no rows are returned, the table is created, but it is left empty. Query (50) creates a new table named **myevents** by using the results of a SELECT to include only events within the specified latitude and longitude range.

(50) SQL> **create table** myevents as
 select *
 from origin
 where lat between 35.0 and 40.0
 and lon between 50.0 and 75.0;
 Table created.

SQL> select orid, lat, lon, time, depth, mb
 from myevents;

ORID	LAT	LON	TIME	DEPTH	MB
3679	36.8840	73.3430	636714964.102	19.7271	4.07
3512	36.8908	73.4689	636732572.863	9.1589	3.98

SQL>

Use the DROP TABLE command to remove a table.

SQL> drop table *name*;

CHANGING TABLE CONTENTS

The contents of a table may be altered by inserting new rows, updating the values of columns in rows, and deleting rows.

INSERT

The INSERT INTO command inserts new rows in a table. The new row or rows may either be defined explicitly or as the result of a query. An explicit insertion uses a VALUES statement as follows:

```
SQL> insert into name
      values (value1 [, value2 [, ...]]);
```

SQL ▼
Commands

Character strings in the VALUES statement must be in single quotes. The values do not have to be in the order specified by the table as long as the column names into which they are being placed are defined:

```
SQL> insert into name
      value1_column_name[, value2_column_name [, ...]])
      values (value1 [, value2 [, ...]]);
```

Use the following syntax to insert rows resulting from a query into a table:

```
SQL> insert into name
      (column1 [, column2 [, ...]])
      select ...
      from ...;
```

Query (51) inserts a new row into the **myevents** table.

```
(51) SQL> insert into myevents
      select *
      from   origin
      where  lat between 30.0 and 35.0
      and    lon between 50.0 and 75.0;
1 row created.
```

```
SQL> select orid, lat, lon, time, depth, mb
      from myevents;
```

ORID	LAT	LON	TIME	DEPTH	MB
3679	36.8840	73.3430	636714964.102	19.7271	4.07
3512	36.8908	73.4689	636732572.863	9.1589	3.98
3514	32.9234	74.9347	636734589.275	12.9065	3.49

```
SQL>
```

DELETE

The DELETE FROM command deletes rows from a specified table. A WHERE clause is usually used in conjunction with DELETE FROM, because without it, the command deletes all of the rows of the table.

```
SQL> delete from name
      where ...;
```

Query (52) deletes the new row from the **myevents** table.

```
(52) SQL> delete from myevents
      where    lat between 30.0 and 35.0
      and      lon between 50.0 and 75.0;
1 row deleted.

SQL> select orid, lat, lon, time, depth, mb
      from myevents;
```

ORID	LAT	LON	TIME	DEPTH	MB
3679	36.8840	73.3430	636714964.102	19.7271	4.07
3512	36.8908	73.4689	636732572.863	9.1589	3.98

```
SQL>
```

UPDATE

The UPDATE command changes specific table entries. The SET command lists the columns that will be changed. The WHERE clause can limit the number of rows affected by the UPDATE.

```
SQL> update name set column=value [, column=value [, ...]]
      where ...;
```

An embedded SELECT command also updates table entries:

```
SQL> update name set column = (
      select ...
      from ...
    );
```

SQL ▼ Commands

To change more than one column, enclose the columns to be changed in parentheses after the SET command:

```
SQL> update name set (column [, column [, ...]]) = (
      select ...
      from ...
    );
```

When changing more than one column, match the number of columns in the SET list with the number of columns returned by the SELECT command.

Query (53) updates the magnitude (*mb*) field of the **myevents** table by using a conversion formula to increase the magnitudes by 5 percent.

```
(53) SQL> update myevents set mb=mb*1.05;
      2 rows updated.
```

```
SQL> select orid, lat, lon, time, depth, mb
      from myevents;
```

ORID	LAT	LON	TIME	DEPTH	MB
3679	36.8840	73.3430	636714964.102	19.7271	4.28
3512	36.8908	73.4689	636732572.863	9.1589	4.18

```
SQL>
```

ROLLBACK and COMMIT

The results of the INSERT INTO, DELETE FROM, or UPDATE commands are not made permanent until you COMMIT them. Until the command is committed, only the user who made the changes can view the results of the changes. The COMMIT command is as follows:

```
SQL> commit;
```

If you discover a mistake prior to committing the changes, you can reverse the changes by using the ROLLBACK command. After using COMMIT, however, you cannot ROLLBACK the changes.

Query (54) returns the magnitude (*mb*) field of the **myevents** table to the original values and removes this table from the database.

(54) SQL> **rollback;**
Rollback complete.

SQL> select orid, lat, lon, time, depth, mb
from myevents;

ORID	LAT	LON	TIME	DEPTH	MB
3679	36.8840	73.3430	636714964.102	19.7271	4.07
3512	36.8908	73.4689	636732572.863	9.1589	3.98

SQL> drop table myevents;

Table dropped.

SQL>

SQL*Plus Extensions

This chapter describes the SQL*Plus extensions used at the IDC and includes the following topics:

- Query Buffer
- Character Functions
- Number Functions
- Manipulating Dates and Times

SQL*Plus Extensions and Functions

The SQL standard does not include an interactive query interface for entering and modifying queries. Additionally, a date data type is not included in the SQL standard; therefore, any definition, storage, or manipulation of dates is a vendor-specific extension to SQL. The data dictionary, which stores information about objects in the database, also varies from one vendor to the next.

This chapter addresses a few ORACLE extensions to the SQL standard and describes many of the functions available for manipulating data.

QUERY BUFFER

SQL*Plus maintains an active query buffer containing the last query run. This section introduces a few of the many commands available for working with the query buffer. Complete information about query buffers is available in [Koc97].

One common use of the query buffer is to edit and resubmit a query. For example, Query (55) selects the latest time from the **arrival** table.

```
(55)      SQL>  select    max(time)
           from      arrival;
```

```

MAX(TIME)
-----
636739132
```

Query (56) uses the LIST command to print the query buffer to the screen.

```
(56)  SQL> list
      1  select  max(time)
      2*  from  arrival
```

The active line of the query buffer is tagged with an asterisk. You can specify what line should be active by entering the line number at the SQL prompt. For example, Query (57) makes the first line active.

```
(57)  SQL> 1
      1*  select  max(time)
```

The CHANGE command edits text on the active line. Query (58) changes max to min.

```
(58)  SQL> change/max/min
      1*  select  min(time)
```

The RUN or / command reruns the query. Query (59) uses / to rerun Query (58).

```
(59)  SQL> /

      MIN(TIME)
-----
      636710455
```

The FORMAT command controls the output displayed for the named COLUMN. For example, Query (59) returned the minimum time from **arrival** but did not include any decimal digits. Query (60) uses the nine numeric template to format the output to include three decimal digits and then reruns the query.

SQL*Plus ▼
Extensions

```
(60)  SQL>  column min(time) format 999999999.999
      SQL>  /
```

```

      MIN(TIME)
-----
      636710455.898
SQL>
```

Column headings may be reset. Query (61) makes the first line of the query buffer active and then appends text to the end of the line using the APPEND command. This query names the column heading *min_time*.

```
(61)  SQL>  1
      1* select  min(time)
      SQL>  append min_time
      1* select min(time)min_time
      SQL>  list
      1  select min(time)min_time
      2* from arrival
      SQL>  /
```

```

      MIN_TIME
-----
      636710455
SQL>
```

Query (62) formats column *min_time* to include three decimal digits.

```
(62)  SQL>  column min_time format 999999999.999
      SQL>  /
```

```

      MIN_TIME
-----
      636710455.898
SQL>
```

You can edit the query buffer through a UNIX editor, such as *vi*. Query (63) first uses the DEFINE command to set the SQL*Plus `_EDITOR` to `/usr/ucb/vi` and then uses the EDIT command to invoke the editor (*vi*) on the query buffer. The boxed information represents the commands entered in *vi*.

```
(63)  SQL> define _editor=/usr/ucb/vi
      SQL> edit
```

```
select min(time)min_time
from arrival
/
~
~
~
~
~
```

At this point, use *vi* commands to edit the query. The following commands change all occurrences of *min* back to *max*:

```
select max(time)max_time
from arrival
/
~
~
~
~
~
~
```

Upon saving the changes with the *vi* commands `ZZ` or `:wq`, the changed query is displayed followed by the `SQL>` prompt. Query (64) uses the backslash (`/`) command to rerun the query.

SQL*Plus ▼
Extensions

```
(64)  1      select max(time)max_time
      2*      from arrival
      SQL>  /
```

```
      MAX(TIME)
-----
      636739132

      SQL>
```

Query (65) uses the SAVE command to write the query buffer to a UNIX file called `max_arrival_time` and then runs it from the version in the file either with the START command or by prepending the filename with the @ character.

```
(65)  SQL> save  max_arrival_time
      Created file max_arrival_time
```

```
      SQL> start  max_arrival_time

      MAX(TIME)
-----
      636739132

      SQL>
```

The SQL*Plus HOST command allows a UNIX command to run inside SQL*Plus. The file created with the SAVE command may be listed with the UNIX command `ls`. SQL*Plus automatically adds the `.sql` extension to the file name.

```
(66)  SQL> host ls -l max_arrival_time
      max_arrival_time not found

      SQL> host ls -l max_arrival_time.sql
      -rw-rw-r--  1 demo          40 Aug 28 15:36 max_arrival_time.sql

      SQL>
```

The query buffer processes one command at a time. A group of commands can be put in a file and run with the START command as shown in Query (65). Each command in the file must be terminated with ; or /. Query (67) edits the file `max_arrival_time.sql` to include a command that formats the *output* column. The query buffer is still being edited through the UNIX editor, *vi*.

(67) SQL> edit max_arrival_time

```
select max(time)max_time
from arrival
/
~
~
~
~
~
~
```

The FORMAT command is added for the *max_time* column:

```
column max_time format 999999999.999;
select max(time) max_time
from arrival
/
~
~
~
~
~
~
```

SQL> start max_arrival_time

```
MAX_TIME
-----
636739132.406
```

SQL>

In Query (68), the CLEAR BUFFER command is used to clear the query buffer.

```
(68) SQL> clear buffer
buffer cleared
```

You can save user-defined SQL*Plus startup commands in a file called `login.sql`. When invoked, SQL*Plus executes an installation command file that was created by your ORACLE DBA. Next it searches for `login.sql`, first in the current working directory, then in a search path specified in the `SQLPATH` environmental variable. The following UNIX command shows how to set a search path for SQL*Plus.

```
setenv SQLPATH /usr/local/scripts:/myhome:/myhome
/oracle/scripts
```

By default, query results are displayed interactively on the computer screen. The information that appears on the screen can be written or spooled to a file through the SPOOL command:

```
spool filename
```

This command deletes the contents of the *filename* and writes the information that appears on the screen until the SPOOL OFF command is given:

```
spool off
```

Table 4 summarizes the query buffer commands described in this section. Commands that can be abbreviated to the first character are noted with parentheses.

TABLE 4: SQL*PLUS QUERY BUFFER COMMANDS

Command	Description
(A)PPEND	appends text to the active line
(C)HANGE	changes the contents of the query buffer
CLEAR BUFFER	clears the query buffer
COLUMN <i>column_name</i> FORMAT <i>template</i>	sets the display format
DEFINE _EDITOR= <i>/usr/ucb/vi</i>	sets the editor to <i>vi</i>

TABLE 4: SQL*PLUS QUERY BUFFER COMMANDS (CONTINUED)

Command	Description
EDIT	edits the query buffer
EDIT <i>filename</i>	edits the named file
HOST <i>command</i>	runs a UNIX command
(L)IST	lists the contents of the query buffer
(R)UN	runs the query in the buffer
/	runs the query in the buffer
SAVE <i>filename</i>	writes the query buffer to <i>filename</i>
SPOOL <i>filename</i>	turns on the spooling to <i>filename</i>
SPOOL OFF	turns off spooling
START <i>filename</i>	runs the commands in the named file
@ <i>filename</i>	runs the commands in the named file

CHARACTER FUNCTIONS

Character functions manipulate strings of characters. Some of the functions modify the original strings, and some provide information about the strings (such as where certain sequences of characters appear in a string). Table 5 lists many of the character functions available through ORACLE. Most of these functions are not described in this tutorial; see [Koc97] for descriptions of their use.

TABLE 5: CHARACTER FUNCTIONS

Function	Definition
<i>string1</i> <i>string2</i>	concatenates the two strings
ASCII(<i>string</i>)	returns the ASCII value of the first character of <i>string</i>
CHR(<i>ASCII value</i>)	returns the printable character that <i>ASCII value</i> represents

TABLE 5: CHARACTER FUNCTIONS (CONTINUED)

Function	Definition
CONCAT(<i>string1</i> , <i>string2</i>)	concatenates the two strings
INITCAP(<i>string</i>)	returns <i>string</i> with initial characters in upper case
INSTR(<i>string</i> , <i>set</i> [, <i>start</i> [, <i>occurrence</i>]])	returns the location of the <i>occurrence</i> of <i>set</i> characters in <i>string</i> ; the search begins at character number <i>start</i> .
LENGTH(<i>string</i>)	returns the number of characters in <i>string</i>
LOWER(<i>string</i>)	returns <i>string</i> in lower case
LPAD(<i>string</i> , <i>length</i> [, <i>'set'</i>])	pads <i>string</i> on the left with the optional <i>set</i> characters (blank is default) to make <i>string</i> <i>length</i> -characters long
LTRIM(<i>string</i> [, <i>'set'</i>])	trims <i>string</i> on the left of the optional <i>set</i> characters (blank is default)
RPAD(<i>string</i> , <i>length</i> [, <i>'set'</i>])	pads <i>string</i> on the right with the optional <i>set</i> characters (blank is default) to make <i>string</i> <i>length</i> -characters long
RTRIM(<i>string</i> [, <i>'set'</i>])	trims <i>string</i> on the right of the optional <i>set</i> characters (blank is default)
SUBSTR(<i>string</i> , <i>start</i> [, <i>count</i>])	returns <i>count</i> letters from <i>string</i> beginning at character number <i>start</i>
UPPER(<i>string</i>)	returns <i>string</i> in upper case
VALUE(<i>string</i>)	returns a mathematical value for <i>string</i> , assuming that <i>string</i> is a set of characters representing numbers

NUMBER FUNCTIONS

ORACLE includes three types of number functions: single-value functions, group-value functions, and list functions. In addition to these, the IDC has added functions that apply particularly to monitoring. Many of these functions and their definitions are listed in Tables 6 through 8. Group-value functions are described in Table 3 on page 22. Examples of the IDC functions are provided after the tables.

TABLE 6: ORACLE SINGLE-VALUE FUNCTIONS

Function	Definition
$value1 + value2$	addition
$value1 - value2$	subtraction
$value1 * value2$	multiplication
$value1 / value2$	division
ABS(<i>value</i>)	absolute value of <i>value</i>
ACOS(<i>value</i>)	arc cosine of <i>value</i> in radians
ASIN(<i>value</i>)	arc sine of <i>value</i> in radians
ATAN(<i>value</i>)	arc tangent of <i>value</i> in radians
ATAN2(<i>value1</i> , <i>value2</i>)	arc tangent of <i>value1</i> / <i>value2</i> in radians
CEIL(<i>value</i>)	<i>value</i> is a decimal number rounded upwards to an integer
COS(<i>value</i>)	cosine of <i>value</i>
COSH(<i>value</i>)	hyperbolic cosine of <i>value</i>
EXP(<i>value</i>)	e raised to <i>value</i> power
FLOOR(<i>value</i>)	<i>value</i> is a decimal number rounded downwards to an integer
LOG(<i>base</i> , <i>value</i>)	logarithm to base <i>base</i> of <i>value</i>
MOD(<i>value</i> , <i>divisor</i>)	modulus (remainder) of <i>value</i> / <i>divisor</i>
NVL(<i>value</i> , <i>substitute</i>)	replaces <i>value</i> with <i>substitute</i> , if <i>substitute</i> is NULL

TABLE 6: ORACLE SINGLE-VALUE FUNCTIONS (CONTINUED)

Function	Definition
POWER(<i>value</i> , <i>exponent</i>)	<i>value</i> to <i>exponent</i> power
ROUND(<i>value</i> , <i>precision</i>)	rounds <i>value</i> to <i>precision</i> decimal places
SIGN(<i>value</i>)	−1 if <i>value</i> < 0 and +1 otherwise
SIN(<i>value</i>)	sine of <i>value</i>
SINH(<i>value</i>)	hyperbolic sine of <i>value</i>
SQRT(<i>value</i>)	square root of <i>value</i>
TAN(<i>value</i>)	tangent of <i>value</i>
TANH(<i>value</i>)	hyperbolic tangent of <i>value</i>
TRUNC(<i>value</i>)	truncates <i>value</i> to an integer without rounding

TABLE 7: ORACLE LIST FUNCTIONS

Function	Definition
GREATEST(<i>value1</i> , <i>value2</i> ,...)	greatest of <i>value1</i> , <i>value2</i> ,...
LEAST(<i>value1</i> , <i>value2</i> ,...)	least of <i>value1</i> , <i>value2</i> ,...

TABLE 8: IDC FUNCTIONS

Function	Definition
AZIMUTH(<i>lat1</i> , <i>lon1</i> , <i>lat2</i> , <i>lon2</i> , <i>back</i>)	returns azimuth in degrees, clockwise from north of point 2 (<i>lat2</i> , <i>lon2</i>) as seen from point 1 (<i>lat1</i> , <i>lon1</i>) when <i>back</i> is 0, and returns backazimuth when <i>back</i> is 1
DEGACOS(<i>value</i>)	arc cosine of <i>value</i> in degrees
DEGASIN(<i>value</i>)	arc sine of <i>value</i> in degrees
DEGATAN(<i>value</i>)	arc tangent of <i>value</i> in degrees

TABLE 8: IDC FUNCTIONS (CONTINUED)

Function	Definition
DEGATAN2(<i>value1</i> , <i>value2</i>)	arc tangent of <i>value1/value2</i> in degrees
DEG_DISTANCE(<i>lat1</i> , <i>lon1</i> , <i>lat2</i> , <i>lon2</i>)	returns the distance in degrees between point 1 (<i>lat1</i> , <i>lon1</i>) and point 2 (<i>lat2</i> , <i>lon2</i>) on the earth's surface
KM_DISTANCE(<i>lat1</i> , <i>lon1</i> , <i>lat2</i> , <i>lon2</i>)	returns the distance in kilometers between point 1 (<i>lat1</i> , <i>lon1</i>) and point 2 (<i>lat2</i> , <i>lon2</i>) on the earth's surface

The **assoc** table provides the azimuth, backazimuth, and distance calculated from the origin to the stations of every phase associated with an origin: *seaz* (station-to-event azimuth), *esaz* (event-to-station azimuth), and *delta*. For stations that do not have associated phases, however, the distance and azimuth must be calculated. Query (69) uses the IDC functions to obtain the azimuth, backazimuth, distance in degrees, and distance in kilometers from station EKA to origin 3679.

```
(69) SQL> select  azimuth(s.lat, s.lon, o.lat, o.lon, 0) azimuth,
                azimuth(s.lat, s.lon, o.lat, o.lon, 1) back_azimuth,
                deg_distance(s.lat, s.lon, o.lat, o.lon) deg_distance,
                km_distance(s.lat, s.lon, o.lat, o.lon) km_distance
          from    origin o, site s
         where    o.orid=3679
         and      s.sta='EKA';
```

```
          AZIMUTH BACK_AZIMUTH DEG_DISTANCE KM_DISTANCE
-----
76.4222298    316.270036    53.141544    5909.07614

SQL>
```

MANIPULATING DATES AND TIMES

A *date* field may be used in any SQL clause that allows a number or character field. ORACLE *date* fields are stored in an internal format. All database operations must convert to and from this internal format using the TO_CHAR() and TO_DATE() functions. TO_CHAR() converts a field from ORACLE's internal format to a character string. TO_DATE() converts a field from a character string or number to ORACLE's internal date format. ORACLE date and time functions are summarized in Table 9, and some of the date formats are listed in Table 10.

All IDC database dates use Greenwich Mean Time (GMT).

TABLE 9: ORACLE DATE AND TIME FUNCTIONS

Function	Definition
TO_CHAR(<i>date</i> , ' <i>format</i> ')	converts an ORACLE <i>date</i> into a string using <i>format</i>
TO_DATE(<i>string</i> , ' <i>format</i> ')	converts <i>string</i> into an ORACLE date using <i>format</i>

TABLE 10: ORACLE DATE AND TIME FORMATS

Format	Definition	Example
DD	day of month number	01
DY	three-character day of week in capital letters	FRI
Dy	three-character day of week with initial letter capitalized	Fri
dy	three-character day of week in lower case letters	fri
DAY	day of week in capital letters	FRIDAY
Day	day of week with initial letter capitalized	Friday
day	day of week in lower case letters	friday
DDD	day of year	365
MI	minute of hour	30
MM	month number of year	06

TABLE 10: ORACLE DATE AND TIME FORMATS (CONTINUED)

Format	Definition	Example
MON	three-character month name in capital letters	JUN
Mon	three-character month name with initial letter capitalized	Jun
mon	three-character month name in lower case letters	jun
MONTH	month name in capital letters	JUNE
Month	month name with initial letter capitalized	June
month	month name in lower case letters	june
SS	second of minute	04
YYYY	four-character year number	1998
YY	last two characters of the year number	98

In the following examples, dates are represented as two-digit years (the YY format) for display purposes. Internally, ORACLE represents dates as four-digit years (the YYYY format). The following data template is used for the examples:

```
'MM/DD/YY HH24:MI:SS'
```

January 15, 1970 at 11 p.m. is formatted as follows:

```
01/15/70 23:00:00
```

The date template must be enclosed by single quotes. The characters / and :, and the space separating the date from the time are optional string characters. The following template omits those characters:

```
'MMDDYYHH24MISS'
```

January 15, 1970 at 11 p.m. would then be output as follows:

```
011570230000
```

The HH24 format instructs ORACLE to output dates on a 24-hour clock.

Selecting Dates

Given the name of the field to convert and a display format, the TO_CHAR() function converts the field from ORACLE's internal format to a character string. Query (70) selects *sysdate*, an ORACLE internal column containing the current date and time, from the table **dual**.

```
(70)  SQL> select  sysdate
        from      dual;
```

```
SYSDATE
-----
23-SEP-97
SQL>
```

By default, ORACLE returns only the date portion. Query (71) reformats the output to change the date format and include the time.

```
(71)  SQL> select  to_char(sysdate,'MM/DD/YY HH24:MI:SS')
        from      dual;
```

```
TO_CHAR(SYSDATE,'MM/DD/YYHH24:MI:SS')
-----
09/23/97 13:44:06
SQL>
```

Query (72) changes the column heading to *now* and makes it 17 characters wide with the A (alphanumeric) display format.

```
(72)  SQL> column  now format A17
SQL> select  to_char(sysdate,'MM/DD/YY HH24:MI:SS') now
        from      dual;
```

```
NOW
-----
09/23/97 13:45:38
SQL>
```

Query (73) DEFINES *now* and *lddate* to be substitution variables. The query also renames the columns *now* and *lddate*, respectively, and so requires quotes around the definition. The character & substitutes the defined string in the place of *&now* or *&lddate*. These definitions are included in the global login startup file at the IDC and can be used in any query. The & operator prints the value of a substitution variable.

```
(73)  SQL> define    now="to_char(sysdate,'MM/DD/YY HH24:MI:SS') now"
      SQL> column    now format A17
      SQL> define    lddate="to_char(lldate,'MM/DD/YY HH24:MI:SS') lddate"
      SQL> column    lddate format A17
      SQL> select    &lddate
           from      origin
           where     orid=3509;
```

```
LDDATE
-----
03/12/90 16:15:51
SQL>
```

Converting between Epoch Times and Dates

All times in the PIDC and IDC databases are recorded in epoch time. This time counts, to millisecond accuracy, the number of seconds since midnight January 1, 1970. Query (74) displays the time for *wfid* 5532.

```
(74)  SQL> column    time format 999999999.999
      SQL> select    wfid, time
           from      wfdisc
           where     wfid=5532;
```

```
WFID      TIME
-----
5532      636737307.566
SQL>
```

Epoch time has little meaning to people, so substitution variables are included in the IDC global startup file to convert epoch times to human-readable times.

In Query (74) *etoh* (epoch-to-human) combines ORACLE's TO_DATE and TO_CHAR functions to convert epoch time to a human-readable date. The *time* field is divided by 86400 (the number of seconds in a day) to calculate the number of days, hours, minutes, and seconds it represents. Fractional seconds are dropped. That result is added to January 1, 1970, to provide the date and time. Because January 1, 1970 is represented by the character string '01/01/1970' and *time* is a floating point field, the TO_DATE function is used to convert both the character string and *time* to the DATE data type so that they may be added. The TO_CHAR function is used to display the resulting date as a character string. The column heading is named *etoh* and made 17 characters wide.

```
(75)  SQL>  define   etoh="to_char(to_date('01/01/1970','MM/DD/YYYY')+ -
        (time/86400),'MM/DD/YY HH24:MI:SS') etoh"
SQL>  column   etoh format A17
SQL>  select    time, &etoh
        from      wfdisc
        where     wfid=5532;

            TIME ETOH
-----
636737307.566 03/06/90 15:28:28

SQL>
```

In Query (75) the substitution variable for *etoh* is too long to fit on a single line. Although SELECT statements may continue freely across lines, as shown in Query (73), if a DEFINE spans more than one line, a dash (–) must be used to alert ORACLE that the definition continues onto the next line.

Etoh substitutes a human readable version of the *time* column in a query. If the query joins two tables that each contain a *time* column, then using *&etoh* will cause an error because ORACLE cannot identify the specific *time* column to be converted. Two substitutions, *w_etoh* and *o_etoh*, provide specific time conversions for the **wfdisc** and **origin** tables of the IDC database. When using these sub-

stitutions, correlation names may not be used for the **origin** or **wfdisc** tables. Query (76) defines *w_etoh* and *o_etoh* and uses *o_etoh* to obtain the time of an origin in a join with the **arrival** table.

```
(76) SQL> define w_etoh="to_char(to_date('01/01/1970','MM/DD/YYYY')+ -
      (wfdisc.time/86400),'MM/DD/YY HH24:MI:SS') w_etoh"
SQL> column w_etoh format A17
SQL> define o_etoh="to_char(to_date('01/01/1970','MM/DD/YYYY')+ -
      (origin.time/86400),'MM/DD/YY HH24:MI:SS') o_etoh"
SQL> column o_etoh format A17
SQL> select lat, lon, &o_etoh, phase, azimuth, slow
      from assoc a, arrival ar, origin
      where a.orid=3509
      and origin.orid=3509
      and a.arid=ar.arid;
```

LAT	LON	O_ETOH	PHASE	AZIMUTH	SLOW
-20.4274	-67.2272	03/06/90 12:57:43	PKP	276	1.90
-20.4274	-67.2272	03/06/90 12:57:43	LR	-1	-1.00
-20.4274	-67.2272	03/06/90 12:57:43	P	131	4.30

SQL>

Etoh3 is the decimal version of *etoh* with time displayed to three decimal places. The decimals (obtained from subtracting the truncated time value from the complete time value) are concatenated onto the end of the standard *etoh* definition. *Dtime* is the same as *etoh* except that default null times in the database (9999999999.999) are returned as null date strings. Query (77) defines both *etoh3* and *dtime* and provides a result for *etoh3*.

SQL*Plus ▼
Extensions

```
(77)  SQL> define etoh3="to_char(to_date('01/01/1970','MM/DD/YYYY')+ -
      (time/86400),'MM/DD/YY HH24:MI:SS' || -
      ltrim(to_char(time-trunc(time),'.099')) etoh3"
SQL> column etoh3 format A21
SQL> define dtime="decode(time,-9999999999.999,'NO TIME PROVIDED ', -
      to_char(to_date('01/01/1970','MM/DD/YYYY')+ -
      (time/86400),'MM/DD/YY HH24:MI:SS')) dtime"
SQL> column dtime format A17
SQL> select time, &etoh3
      from wfdisc
      where wfid=5532;

              TIME ETOH3
-----
636737307.566 03/06/90 15:28:28.566

SQL>
```

Improving Query Performance

This chapter describes how to improve query performance and includes the following topics:

- Using Indexed Columns
- Listing Most Restrictive Tables Last
- Using IN Versus EXISTS

Improving Query Performance

Many factors, including system load and data set size, affect query performance. The more a system is loaded, the slower the results are returned. Additionally, all queries perform well on small data sets such as the *geodemo* data set. But a query that runs quickly on a small data set may have significant performance problems on a large data set.

Although you may not be able to control system load or data set size, you can control the syntax and wording of SQL queries, which can improve performance dramatically. In particular, the following tips will help you improve query performance:

- Reference indexed columns in the WHERE clause.
- List the most restrictive table last in the FROM clause.
- Use the size of the inner query and indexing to decide if IN or EXISTS will perform better.

This section explains each of these tips in greater detail. Most of the queries in this section were run on a data set containing eight weeks of data, so output will not match the same query run on the *geodemo* data set. The **arrival** and **assoc** tables in the eight-week data set contain 46,856 and 4,396 records respectively.

The ORACLE command SET TIMING ON is useful for comparing the performance time of different versions of the same query. Query (78) shows an elapsed time of .05 seconds.

```
(78)  SQL> set timing on;
      SQL> select   orid, count(arid)
        from      assoc
        where      orid < 3510
        group by   orid
        having     count(arid) > 2
        order by   2 desc;
```

ORID	COUNT(ARID)
3506	12
3508	8
3503	6
3504	4
3509	3

Elapsed: 00:00:00.05

SQL>

USING INDEXED COLUMNS

To optimize query performance, the DBA will define indexes on columns. An index tracks data the way a card catalog at the library tracks books. Given the name of a book, the card catalog provides its location. A database index provides a similar mechanism for retrieving data and eliminates the time-consuming search of the whole table. In IDC databases, indexes can fill as much disk space as the data.

Indexed columns should be referenced in the WHERE clause. For example, given a query on the **wfdisc** table for a station and channel, the *sta* and *chan* columns must be indexed for the query to run efficiently. ORACLE databases contain data dictionary tables, **user_ind_columns** and **all_ind_columns**, which contain the indexed columns. **User_ind_columns** contains only the indexes for tables in the current account, and **all_ind_columns** contains the indexes for all tables in the database instance (table names and owners are stored in uppercase).

Improving Query Performance ▼

Query (79) shows that the **wfdisc** table of the *geodemo* instance has three indices: *wfidx*, *wfstax*, and *wftimex*. *wfidx* contains a single column, *wfstax* contains three columns, and *wftimex* contains two columns. An index containing more than one column is called a concatenated index. The index name should never be used in a query because the ORACLE optimizer decides which index to use.

```
(79)  SQL> column index_name format A15
      SQL> column table_name format A15
      SQL> column column_name format A15
      SQL> select index_name, table_name, column_name, column_position,
                column_length
      from all_ind_columns
      where table_name='WFDISC'
      and   table_owner='GEODEMO';
```

INDEX_NAME	TABLE_NAME	COLUMN_NAME	COLUMN_POSITION	COL_LEN
WFIDX	WFDISC	WFID	1	22
WFSTAX	WFDISC	STA	1	6
WFSTAX	WFDISC	CHAN	2	8
WFSTAX	WFDISC	TIME	3	22
WFTIMEX	WFDISC	TIME	1	22
WFTIMEX	WFDISC	ENDTIME	2	22

```
SQL>
```

Sometimes ORACLE ignores the indexed files and scans the whole table. The following list provides common reasons for these occurrences and strategies for improving query performance through the use of indexed columns:

1. An index is ignored if the left-most part of a concatenated index is not referenced.

If the index is made up of more than one column, include the first (left-most) column in the query. You can omit columns to the right.

The following query finds waveform data for all short-period instruments:

```
SQL> select *
      from wfdisc
      where chan = 'se';
```

To avoid a full table scan, specify the array of interest:

```
SQL> select *
      from wfdisc
      where sta like 'AR%' and chan = 'se';
```

2. An index is ignored if a wildcard is the first character of the constant.

The following query causes a full table scan:

```
SQL> select *
      from wfdisc
      where sta like '%R';
```

3. An index is ignored if the column is modified by a function or an arithmetic operation, as shown in the following query:

```
SQL> select *
      from wfdisc
      where sta = 'NRA0'
      and substr(chan,1,1) = 's';
```

By avoiding use of the SUBSTR function, an index on *chan* is used:

```
SQL> select *
      from wfdisc
      where sta = 'NRA0'
      and chan like 's%';
```

4. Performance suffers when functions are used. If a predicate mixes datatypes, avoid using functions on the column, as shown in the following query:

```
SQL> select wfid from wfdisc
      where to_char(to_date('01/01/1970','MM/DD/YYYY')+
                    (time/86400),'MM/DD/YY HH24:MI:SS')
            between '03/06/90' and '03/07/90';
```

Instead, write the query as follows:

```
SQL> select wfid from wfdisc
      where time between
            (to_date('03/06/90','MM/DD/YY')-
             to_date('70/01/01','YY/MM/DD'))*86400
            (to_date('03/07/90','MM/DD/YY')-
             to_date('70/01/01','YY/MM/DD'))*86400;
```

5. An index will not be used if a query contains != or NOT, as shown in the following query:

```
SQL> select *
      from arrival
      where sta != 'ARC';
```

6. Even if an index is used, restrict the range if at all possible to avoid a wide scan, as shown in the following query:

```
SQL> select *
      from arrival
      where time <= 636720000 or time >= 636730000;
```

Instead, restrict the range as follows:

```
SQL> select *
      from arrival
      where (time between 636720000-2400 and 636720000)
            or (time between 636730000 and 636730000+2400);
```

Unless 75 to 80 percent of the table is eliminated by a WHERE clause, the use of an index will add more overhead than a table scan. These strategies may be used to deliberately disable indices.

Finally, ORACLE will use five indices at most. Disable indices on other columns to control which columns the optimizer considers as a candidate for an index.

**LISTING MOST RESTRICTIVE
TABLES LAST**

ORACLE uses a query optimizer that attempts to determine the driving table for joins. ORACLE will use the table order, however, when querying tables that are both indexed, if no other information determines the driving table. When using the table order, the tables in the FROM clause are processed from right to left. The last table in the FROM clause is the driving table, so it should contain the fewest rows for ORACLE to process initially. For queries in which a WHERE clause condition returns very few rows from one of the tables, that table should be listed last. If full table scans are likely, the table with the fewest rows should be listed last. Query (80) and Query (81) were run on a data set containing eight weeks of data (**arrival** contains 46,856 records and **assoc** contains 4,396 records). Query (80) took over three minutes to complete, and Query (81) took only 40 seconds. The only difference between the two queries is the order of the tables in the FROM clause.

```
(80) SQL> set timing on;
SQL> select  ac.phase, count(ac.arid), avg(ar.slow)
      from    assoc ac, arrival ar
      where   ac.arid=ar.arid
      and     ar.slow > 0.0
      and     ac.phase in ('Pn', 'Pg', 'Sn', 'Lg')
      group by ac.phase;
```

PHASE	COUNT(AC.ARID)	AVG(AR.SLOW)
Lg	1549	26.9432021
Pg	417	15.6335971
Pn	1468	13.9206676
Sn	277	23.9224549

Elapsed: 00:03:11.48

SQL>

Improving ▼ Query Performance

```
(81)  SQL> select    ac.phase, count(ac.arid), avg(ar.slow)
        from      arrival ar, assoc ac
        where     ac.arid=ar.arid
        and       ar.slow > 0.0
        and       ac.phase in ('Pn', 'Pg', 'Sn', 'Lg')
        group by  c.phase;
```

PHASE	COUNT(AC.ARID)	AVG(AR.SLOW)
Lg	1549	26.9432021
Pg	417	15.6335971
Pn	1468	13.9206676
Sn	277	23.9224549

Elapsed: 00:00:40.26

SQL>

In Query (80) and Query (81) neither *slow* nor *phase* are indexed, so the driving table is fully scanned. In fact, indices would not help in this case because most **arrival** records will have a slowness greater than 0.0 and most **assoc** records will occur within the specified phases. Given the 75 to 80 percent rule for indices as described in the previous section, scanning the whole table is more efficient than processing an index. The strategy is to limit the full table scan to the smaller of the two tables and extract data from the larger table based on the efficient *arid* join. The **assoc** table contains 4,396 records in this data set and the **arrival** table contains 46,856 records; therefore, **assoc** is a better driving table.

USING IN VERSUS EXISTS

An IN subquery returns the same result as an EXISTS subquery, but ORACLE processes each subquery differently. A subquery has two parts: the outer query and the inner query. The inner query of an IN subquery is executed once, and the results are stored in an unindexed temporary table. Each row in the outer query is then compared to the rows in the temporary table. If the outer row is found in the temporary table, it is returned to the user.

The inner query of an EXISTS subquery is executed repeatedly, once for each row in the outer query. The inner query returns TRUE or FALSE to the outer query. If it returns TRUE, then the row in the outer query is returned to the user.

Sometimes one subquery form will perform better than the other depending on the size of the inner query data set. Query (82) and Query (83) return the earliest associated *arrival* time for a given *orid* in the eight-week data set.

```
(82)  SQL> select  min(time)
        from      arrival
        where      arid in
                (select  arid
                  from    assoc
                  where    orid=105196);
```

```
MIN(TIME)
```

```
-----
623520155
```

```
Elapsed: 00:00:00.35
```

```
SQL>
```

```
(83)  SQL> select  min(time)
        from      arrival
        where      exists
                (select  *
                  from    assoc
                  where    arrival.arid=assoc.arid;
```

```
MIN(TIME)
```

```
-----
623520155
```

```
Elapsed: 00:00:01.46
```

```
SQL>
```

Improving ▼ Query Performance

The difference in performance of Query (82) and Query (83) is negligible but suggests the IN might perform slightly better than the EXISTS. Because the **assoc** table has only three records for *orid* 105196, the temporary table created for Query (82) is quite small, and the inner query for Query (83) is performed only three times.

Query (84) and Query (85), however, show a dramatic difference in performance. Each query returns a count of the unassociated arrivals by station in the eight-week data set.

```
(84)  SQL> select  sta, count(arid)
        from      arrival
        where      arid not in
                (select  arid
                 from      assoc)
        group by  sta;
```

```
STA      COUNT(ARID)
```

```
-----
```

Query killed after 18 hours

```
SQL>
```

```
(85)  SQL> select  sta, count(arid)
        from      arrival
        where      not exists
                (select  arid
                 from      assoc
                 where      assoc.arid=arrival.arid)
        group by  sta;
```

```
STA      COUNT(ARID)
```

```
-----
```

```
ARA0      27193
```

```
NRA0      15270
```

```
Elapsed: 00:02:40.92
```

```
SQL>
```

The inner query in Query (84) does not contain a WHERE clause to narrow the search; the intermediate table contains all 4,396 records from the **assoc** table. Because ORACLE cannot index the temporary table, each of the 46,856 rows in the outer query scans the entire intermediate table. Performance is probably comparable to the 56-hour cartesian product of Query (33). The EXISTS form in Query (85) performs much better than Query (84) because it uses any available index.

In summary, if the temporary table produced by the IN subquery is quite small, the IN could perform better than the EXISTS because it avoids repeated executions of the inner query. If the temporary table produced by the IN subquery is quite large, the EXISTS will perform better because it can use any available index.

Navigating Databases

This chapter describes how databases may be distributed and organized and how to obtain this type of information from a database. The following topics are discussed:

- Instances
- Accounts
- Tables

Navigating Databases

To use the IDC database most efficiently, you should be familiar with the database instances, the accounts that are included in each instance, and the database tables that are included in each account. You should also understand the relationships among the tables within the accounts and the contents of the tables themselves. The latter information is contained in [IDC5.1.1Rev1], and a description of the tables is in preparation. Much of the information is also available in the administrative tables included within the databases and can be accessed through simple queries.

INSTANCES

A database instance is everything needed to run the database (programs, memory, and so on). An instance contains a collection of database accounts with unique names. Using an analogy, consider the database instance as one computer in a network of computers. Several different database instances occur at the IDC, each of which serves a specific purpose. IDC database instances exist for processing of seismic, hydroacoustic, and infrasonic data, for IDC processing of radionuclide data, for archiving of processing results, and for testing new processing schemes.

Different database instances are usually installed on different computers of a computer network to prevent heavy loads on one database, which would affect the performance of the other databases, and as insurance against hardware failure.

ACCOUNTS

A database account is a collection of database tables that have unique names and that are protected by account passwords. In the computer network analogy where the database instances are the computers, database accounts are the user accounts. Each database instance may contain several database accounts. The

database account names must be unique within the database instance, but the same account names may be (and often are) shared by several database instances. In the IDC database instances the accounts are used to separate static information (changed relatively infrequently) from dynamic information (changed regularly). The dynamic accounts represent various stages of data processing.

A list of the database accounts within the database instance is maintained in the **all_catalog** table. Query (86) obtains the names of all accounts.

```
(86)      SQL>  select  distinct owner
           from    all_catalog;
```

```
OWNER
-----
AUTODRM
DFX
IDCEXPORT
IDCLEB
IDCREB
IDCWDB
IDCX
MAP
SEL1
SEL2
SEL3

12 rows selected.

SQL>
```

The DISTINCT command must be used in the query because the **all_catalog** table has one row for every table in the database instance.

TABLES

A table is a set of rows that each contain a specific set of columns. The four types of tables are base, synonym, view, and sequence. A base table contains data. A synonym contains no data itself, but points to a base table, usually in another account. To the user, the synonym looks and behaves like a base table, but queries

Navigating ▼ Databases

to the synonym are actually made to the base table. Synonyms allow sharing of tables between accounts. A view also contains no data itself, but is a collection of pointers to the contents of base tables through execution of a query. The query acts like a window through which information is viewed. As the contents of the base tables change, so do the contents of the view. A sequence is used to track a unique sequence of numbers without having to create a special table.

The **user_catalog** table maintains a list of the tables within the current database account and the table type (TABLE, SYNONYM, VIEW, or SEQUENCE). The **all_catalog** table includes a list of all tables within the current database instance. Query (87) obtains the names of all tables in account *geodemo*.

```
(87)  SQL>  select  table_name
        from      all_catalog
        where      owner='GEODEMO';
```

```
TABLE_NAME
-----
AFFILIATION
ARRIVAL
ASSOC
GREGION
INSTRUMENT
LASTID
NETMAG
NETWORK
NEXTID
ORIGERR
ORIGIN
PATH
REMARK
SENSOR
SITE
SITECHAN
SREGION
```

```
STAMAG
WFDISC
WFTAG

20 rows selected.

SQL>
```

Query (88) uses the DESCRIBE command to obtain a description of the columns of any individual table. No semicolon is required after the DESCRIBE command.

(88) SQL> describe all_catalog

Name	Null?	Type
-----	-----	----
OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLE_TYPE		VARCHAR2(11)

SQL>

Advanced Seismology Queries

This chapter presents queries that are more complex than the previous examples. In some cases, the examples use special features of the IDC schema. Although all sample queries were run on the *geodemo* data set for demonstration purposes, most of them are not interesting unless they are run on a large data set. Queries in this section use ORACLE extensions to the ANSI SQL standard, such as substitution variables, which will not run on other databases. The following topics are included:

- Finding All Events in Arrival Time Windows
- Retrieving Origin Information from Events with Depth Phases
- Estimating Station Residuals
- Calculating Azimuth Resolution
- Performing Linear Regression

Advanced Seismology Queries

FINDING ALL EVENTS IN ARRIVAL TIME WINDOWS

Query (89) finds all events within an arrival time window. It also counts the number of arrivals in the time window.

```
(89)  SQL> define      begin_time = 636725500
      SQL> define      end_time = 636730000
      SQL> select      ac.orid, count(ac.arid)
           from          assoc ac, arrival ar
           where         ac.arid=ar.arid
           and           ar.time between &begin_time and &end_time
           group by     ac.orid;
```

ORID	COUNT(AC.ARID)
3507	2
3508	8
3509	2
3510	3

```
SQL>
```

Query (90) calculates the number of arrivals for each event. Observe the count of *arids* for each *orid*; two arrivals are outside of the original time window, one each for *orids* 3509 and 3510.

```
(90)  SQL> select   orid, count(arid)
        from      assoc
        where     orid in (3507, 3508, 3509, 3510)
        group by  orid;
```

ORID	COUNT(ARID)
3507	2
3508	8
3509	3
3510	4

```
SQL>
```

Query (91) finds arrivals outside the original search time window for *orids* 3509 and 3510.

```
(91)  SQL> select   ac.orid, ar.arid, ar.time
        from      arrival ar, assoc ac
        where     ar.arid=ac.arid
        and       (ar.time < &begin_time or ar.time > &end_time)
        and       (ac.orid between 3509 and 3510);
```

ORID	ARID	TIME
3510	67531	636730145.594
3509	67651	636731411.000

```
SQL>
```

One problem with this query is that the **arrival** table will be fully scanned because data before the *begin_time* and after the *end_time* comprise the majority of the table. Consequently, performance on large data sets will be degraded. You can improve efficiency by limiting the search to a specific duration between the begin and end times. For example, if you know that the travel time is less than 40 minutes, define a duration of 2400 epoch seconds and restate the query as shown in Query (92).

Advanced ▼ Seismology Queries

```
(92)  SQL> define    duration=2400
      SQL> select    ac.orid, ar.arid, ar.time
      from          arrival ar, assoc ac
      where         ar.arid=ac.arid
      and           (ar.time between &begin_time - &duration and &begin_time
                    or ar.time between &end_time and &end_time + &duration)
      and           (ac.orid between 3509 and 3510);
```

ORID	ARID	TIME
3510	67531	636730145.594
3509	67651	636731411.000

SQL>

Query (93) avoids a search for specific *orids* by finding all arrivals outside a given time window that have *orids* within the given time window.

```
(93)  SQL> select    ar.arid, ar.time, ac.orid
      from          arrival ar, assoc ac
      where         ar.arid=ac.arid
      and           (ar.time between &begin_time - &duration and &begin_time
                    or ar.time between &end_time and &end_time + &duration)
      and           ac.orid in
                    (select    ac2.orid
                     from      assoc ac2, arrival ar2
                     where     ac2.arid=ar2.arid
                     and       ar2.time between &begin_time and &end_time
                    );
```

ARID	TIME	ORID
67651	636731411.000	3509
67531	636730145.594	3510

SQL>

RETRIEVING ORIGIN INFORMATION FROM EVENTS WITH DEPTH PHASES

Query (94) demonstrates an EXISTS subquery. Given an origin with a depth greater than 20 km, the **assoc** table is searched for an associated arrival with a *phase* of either 'pP' or 'sP.' If one exists, the origin information is retrieved.

```
(94)  SQL> select  o.orid, o.jdate, o.lat, o.lon, o.depth, o.mb, o.ms, o.ndef
        from      origin o
        where     o.depth > 20.0
        and       exists
            (select  arid
              from    assoc
              where   orid = o.orid
              and     phase in ('pP', 'sP')
            );
```

ORID	JDATE	LAT	LON	DEPTH	MB	MS	NDEF
3506	1990065	12.1327	143.6159	22.1815	4.39	1.91	10
3683	1990065	-10.8992	117.5039	29.6973	5.10	5.39	19
3513	1990065	-17.8557	168.0396	27.5160	4.38	2.7	7

SQL>

ESTIMATING STATION RESIDUALS

Query (95) estimates station residuals by using the phase P associated with origins having at least four defining phases. It returns only those stations that have more than five such phases.

Query (95) uses four basic math and set functions: COUNT, AVG, SQRT, and STDDEV. The GROUP BY clause causes the average to be calculated on a station-by-station basis. The HAVING clause restricts the results to those stations with more than five observations.

Although this query executes properly on the *geodemo* data set, it returns more interesting results when run on larger data sets.

Advanced ▼ Seismology Queries

```
(95)  SQL> select    a.sta, count(a.arid) num,
                    avg(a.timeres) avg_timeres_residual,
                    stddev(a.timeres) stand_timeres_dev
        from        assoc a, origin o
        where       o.orid = a.orid
           and       a.phase = 'P'
           and       o.ndef >= 4
        group by    a.sta
        having       count(a.arid) > 5;
```

STA	NUM	AVG_TIMERES_RESIDUAL	STAND_TIMERES_DEV
ASAR	9	-.46844444	.77499405
ASPA	6	-.97516667	.825876363
WRA	9	-.29188889	1.64798904
YKA	6	.038	.267926109

SQL>

CALCULATING AZIMUTH RESOLUTION

Some IDC database tables contain summary information based on data from fields in other tables. The following queries calculate the *azres* field in the **assoc** table based on the *azimuth* field in the **arrival** table and the *seaz* field in the **assoc** table. By definition:

$$\text{assoc.azres} = \text{arrival.azimuth} - \text{assoc.seaz}$$

The range, however, must be between -180 and 180 degrees. *Azimuth* and *seaz* are defined to be within 0 and 360 degrees, so a simple difference may not fall within the prescribed range.

In Query (96) *azres* is initialized to the NA value.

```
(96)  SQL> update    assoc
        set          azres = -999;
```

120 records updated.

Without the WHERE clause in Query (97) any *arid* with an NA *azimuth* or *seaz* will get reset to a database NULL, overwriting the NA value.

```
(97)  SQL> update    assoc ac
        set         ac.azres =
        (select      max(azimuth)-max(ac.seaz)
        from         arrival
        where        arid=ac.arid
        and          azimuth >= 0.0
        and          ac.seaz >= 0.0)
        where        exists
        (select arid
        from arrival
        where arid=ac.arid
        and azimuth >= 0.0
        and ac.seaz >= 0.0);
```

64 records updated.

If the resulting *azres* is less than -180, 360 must be added, as shown in Query (98).

```
(98)  SQL> update    assoc
        set         azres = azres + 360.0
        where        azres between -360.0 and -180.0;
```

14 records updated.

If *azres* is greater than 180, 360 must be subtracted, as shown in Query (99).

```
(99)  SQL> update    assoc
        set         azres = azres - 360.0
        where        azres > 180.0;
```

5 records updated.

Advanced ▼ Seismology Queries

Query (100) shows the result of the updates performed in Query (96) through Query (99).

```
(100)  SQL>  select    arid, azres
         from      assoc;
```

ARID	AZRES
67437	1.1
67490	-1.3
67454	-4.9
68223	-999.0
67456	-5.9
68225	-999.0
67506	6.0
67507	-999.0
67377	2.9
67378	-0.1
66881	4.8

<many more rows>
120 records selected.
SQL>

In Query (101) these changes are made permanent in the database by the COMMIT command.

```
(101)  SQL>  commit;

Commit complete.
```

PERFORMING LINEAR REGRESSION

This section demonstrates the use of temporary tables by using SQL to perform a linear regression. Assume:

$$Y = A + B * X \pm \text{standard deviation}$$

Suppose P-wave station residuals are correlated with station elevation, then:

$$\text{residual} = A + B * \text{elevation} \pm \text{error}$$

In Query (102) a QR-decomposition method solves the system inside a temporary table.

```
(102)  rem
rem    A line that begins with "rem" is a remark.  ORACLE also allows
rem    comments between /* and */ delimiters.

rem
rem    Ignore error on drop table command--it means that the table does
rem    not exist and may be created.
rem
rem    A FLOAT(24) allows 7.2 decimal digits of precision.
rem
SQL> drop table datamatrix;
drop table datamatrix
*
ERROR at line 1:
ORA-00942: table or view does not exist
SQL> create table datamatrix (
      one float(24),
      X   float(24),
      Y   float(24),
      d12 float(24),
      d13 float(24),
      d23 float(24)
    );
Table created.

rem    Insert X and Y values into a temporary table.
rem    Insert into data matrix elevations and residuals of defining
rem    P arrivals.  Check against NA values for timeres or elev.
rem
rem    This insert...select can be modified for any linear regression.
SQL> insert into datamatrix (one, x, y, d12, d13, d23)
      select 1.0, s.elev, a.timeres, 0.0, 0.0, 0.0
      from   assoc a, site s
      where  s.sta = a.sta
      and    a.timedef = 'd'
```

Advanced ▼ Seismology Queries

```

and      a.phase = 'P'
and      a.timeres > (-999.0 * 0.9999)
and      s.elev > (-999.0 * 0.9999);

```

68 records created.

```

rem
rem  Do the QR decomposition.
rem

```

```

SQL> drop table qr_coeff;
drop table qr_coeff
*

```

```

ERROR at line 1:
ORA-00942: table or view does not exist

```

```

SQL> create table qr_coeff (
      D12 float(24),
      D13 float(24),
      D23 float(24)
    );

```

Table created.

```

rem
rem  In the following insert, all that is really required for this data
rem  set is
rem
rem      select sum(X)/sum(one)
rem
rem  instead of
rem
rem      select sum(one*X)/sum(one*one)
rem
rem  The syntax actually used solves a more general case where the
rem  elements are variable in weight.
rem

```

```

SQL> insert into qr_coeff (D12, D13)
      select  sum(one*X)/sum(one*one),
              sum(one*Y)/sum(one*one)
      from    datamatrix;

```

1 record created.

```

SQL> select  *
      from    qr_coeff;

      D12      D13      D23
-----
.39729118  -.09735294

rem
rem  The double update looks redundant but SQL will not
rem  allow use of subquery results in arithmetic expressions.
rem

SQL> update datamatrix
      set  d12 = (select max(D12) from qr_coeff),
          d13 = (select max(D13) from qr_coeff);
68 records updated.

SQL> update datamatrix
      set  X = X - one * d12,
          Y = Y - one * d13;
68 records updated.

SQL> update qr_coeff
      set  D23 =
          (select  sum(X*Y)/sum(X*X)
           from    datamatrix);

1 record updated.

SQL> update datamatrix
      set  d23 =
          (select  max(D23)
           from    qr_coeff);

68 records updated.

SQL> update datamatrix
      set  Y = Y - X * d23;

68 records updated.

rem
rem  Print coefficients with unbiased variance.
rem

SQL> select  D13 - D12*D23 A, D23 B
      from    qr_coeff;

```

Advanced ▼ Seismology Queries

```

          A          B
-----
.386994551  -1.2191247

SQL>  select  stddev(Y) standev
      from    datamatrix;

      STANDEV
-----
1.06482491

rem
rem  Clean up.
rem

SQL>  drop table qr_coeff;
SQL>  drop table datamatrix;
```

Although this query executes properly on the *geodemo* data set, it returns more interesting results on larger data sets.

Advanced Radionuclide Queries

This chapter presents some complex queries, which are representative of the questions a radionuclide analyst would want answered. In some cases, the examples make subtle use of unique features of the IDC schema. Queries in this section use ORACLE extensions to the ANSI SQL standard, such as substitution variables, which will not run on other databases. The following sample queries are included:

- Searching for Unreviewed FULL Spectra
- Verifying Receipt of Spectra
- Searching for Specific Nuclides
- Determining Concentration Ranges
- Searching for Specific Peaks

Advanced Radionuclide Queries

In this section, multiple queries sometimes are grouped together under a single example to show the steps required for satisfying a particular question. These queries were run against the PIDC Operations database. The SQL queries can all be edited and tailored to the specific needs of the radionuclide specialist.

This chapter uses the following different methods for formatting output:

- Parentheses are used for clarity or to establish precedence.
- Double quotation marks (") are used to assign an alias to a column heading, for example, when the column name is too long for the specified output or when the alias is more self-explanatory than the actual column name.
- BREAK specifies where and how to make format changes to a report. BREAK ON specifies action(s) to be taken when the value of an expression changes. A typical action is to SKIP a line in the report. This expression can involve an alias assigned to a report column in a SELECT statement.
- CLEAR resets or erases the current value or setting for an option (clause). CLEAR BREAKS removes the breaks set by the BREAK command.

The SUBSTR character function, introduced in Table 5 on page 55, is used in the queries of this chapter. SUBSTR returns a portion of the character string, beginning at a specified character and including up to a specified number of characters.

SEARCHING FOR UNREVIEWED FULL SPECTRA

Radionuclide operations staff at the IDC are primarily responsible for the timely review of all incoming particulate and gaseous spectra; therefore, they must know what spectra are waiting to be reviewed. One way to get this information is to query the database. Query (103) searches for FULL spectra that were received after 01 August 1998 and have not yet been reviewed.

```
(103)  SQL> clear breaks
        breaks cleared

        SQL> break on " SITE" skip 1

        SQL> select (station_code) " SITE", (detector_code) " DETECTOR",
                   substr(to_char ((entry_date), 'YYYY-MM-DD HH24:MI'),1,16)
                   " RMS ENTRY DATE", gards_sample_data.sample_id) "SAMPLE#",
                   substr(to_char ((collect_stop), 'YYYY-MM-DD HH24:MI'),1,16)
                   " COLLECTION STOP", auto_category) "CATEGORY",
                   (gards_sample_status.status) "S"
        from      gards_sample_data, gards_sample_status, gards_stations,
                   gards_detectors
        where     gards_sample_data.sample_id = gards_sample_status.sample_id
        and       gards_sample_data.station_id = gards_stations.station_id
        and       gards_sample_data.detector_id = gards_detectors.detector_id
        and       entry_date >= '01-Aug-98'
        and       data_type = 'S'
        and       gards_sample_status.status IN ('A','P')
        and       spectral_qualifier= 'FULL'
        and       collect_stop > collect_start
        order by  station_code, collect_stop,
                   detector_code, acquisition_start;
```

SITE	DETECTOR	RMS ENTRY DATE	SAMPLE#	COLLECTION STOP	CATEGORY	S
CA002	CA002CAA2	1998-08-19 23:12	40277	1998-08-17 23:02	4	P
DE002	DE002IAR3	1998-08-19 11:34	40267	1998-08-03 08:24	3	P
FI001	FI001F09	1998-08-10 20:04	40078	1998-08-03 05:32	3	P
	FI001F09	1998-08-17 10:25	40230	1998-08-10 06:51	3	P
SE001	SE001-ORI	1998-08-20 07:34	40285	1998-08-18 07:00	1	P

```
SQL>
```

VERIFYING RECEIPT OF SPECTRA

Radionuclide Operations staff must also verify that all spectra have been received from a given station, to prevent gaps in the air monitoring records. Query (104) searches the **gards_sample_data** table for all FULL spectra from the FI001 station (station 42) that were collected on or after 01 June 1998.

```
(104)  SQL> clear breaks
        breaks cleared

        SQL> break on " DETECTOR" skip 1

        SQL> select substr((gards_sample_data.site_det_code),1,10)  " DETECTOR",
                   (gards_sample_data.sample_id) "SAMPLE#",
                   substr(to_char ((collect_start), 'YYYY-MM-DD HH24:MI'),1,16)
                   "COLLECTION START",
                   substr(to_char ((collect_stop), 'YYYY-MM-DD HH24:MI'),1,16)
                   " COLLECTION STOP"
        from   guards_sample_data
        where  collect_stop >= '01-Jun-98'
        and    spectral_qualifier = 'FULL'
        and    data_type = 'S'
        and    collect_stop - collect_start > 0
        and    station_id = 42
        order by collect_stop, acquisition_start;
```

DETECTOR	SAMPLE#	COLLECTION START	COLLECTION STOP
-----	-----	-----	-----
FI001F09	34803	1998-05-25 05:47	1998-06-01 05:04
	34934	1998-06-01 05:09	1998-06-08 05:02
	35158	1998-06-08 05:02	1998-06-15 06:04
	35282	1998-06-15 06:07	1998-06-22 05:08
	36885	1998-06-22 05:01	1998-06-29 06:29
	37120	1998-06-29 06:34	1998-07-07 05:58
	37696	1998-07-07 06:00	1998-07-13 06:13
	37697	1998-07-13 06:15	1998-07-20 05:53
	37699	1998-07-20 05:56	1998-07-27 05:23
	40078	1998-07-27 05:25	1998-08-03 05:32
	40230	1998-08-03 05:35	1998-08-10 06:51

SQL>

SEARCHING FOR SPECIFIC NUCLIDES

At some European stations, previous atomic bomb tests and the Chernoybyl nuclear accident in 1986 have resulted in ^{137}Cs being commonly observed in spectra. Query (105) indicates which spectra have ^{137}Cs . This query limits the search to FULL spectra acquired on or after 01 August 1998.

```
(105)  SQL> clear breaks
        breaks cleared

        SQL> break on " DETECTOR" skip 1

        SQL> select substr((gards_sample_data.site_det_code),1,10)  " DETECTOR",
                   substr(to_char ((gards_sample_data.sample_id),
                   '9999999'),1,8) "SAMPLE#",
                   substr(to_char ((collect_stop), 'YYYY-MM-DD HH24:MI'),1,16)
                   " COLLECTION STOP",
                   to_char ((activ_key), '99999999.00') "CONC(uBq/m3)",
                   to_char (((activ_key_err / activ_key) * (100)), '9999.00')
                   "%RELERR"
        from   guards_sample_data, guards_nucl_ided
        where  guards_sample_data.sample_id = guards_nucl_ided.sample_id
        and    guards_nucl_ided.name = 'CS-137'
        and    guards_sample_data.data_type = 'S'
        and    guards_sample_data.acquisition_start >= '01-Aug-98'
        and    guards_sample_data.spectral_qualifier = 'FULL'
        and    activ_key > 0
        order by site_det_code, collect_stop;
```

DETECTOR	SAMPLE#	COLLECTION STOP	CONC(uBq/m3)	%RELERR
DE002IAR3	40267	1998-08-03 08:24	.52	22.23
FI001F09	40078	1998-08-03 05:32	.43	11.96
	40230	1998-08-10 06:51	1.01	12.50

```
SQL>
```

DETERMINING CONCENTRATION RANGES

As seen in the previous output, ^{137}Cs was identified in subsequent FI001 spectra; therefore, the analyst should determine the range of ^{137}Cs concentrations over a given time period, prior to the review of the newly arrived spectra. Query (106) determines the minimum, maximum, average, and standard deviation of ^{137}Cs concentrations at the FI001 station (station 42) measured in FULL spectra collected since 01 June 1998.

```
(106)  SQL> column MINIMUM format 99999.00
        SQL> column MAXIMUM format 99999.00
        SQL> column AVERAGE format 99999.00
        SQL> column STDDEV  format 99999.00
        SQL> clear breaks
        breaks cleared

        SQL> select min(activ_key) "MINIMUM", max(activ_key) "MAXIMUM",
                   avg(activ_key) "AVERAGE", stddev(activ_key) "STDDEV"
        from   gards_sample_data, gards_nucl_ided
        where  gards_sample_data.sample_id = gards_nucl_ided.sample_id
        and    gards_sample_data.station_id = 42
        and    gards_sample_data.spectral_qualifier = 'FULL'
        and    gards_sample_data.data_type = 'S'
        and    gards_sample_data.collect_stop >= '01-Jun-98'
        and    gards_nucl_ided.name = 'CS-137'
        and    activ_key > 0
        and    collect_stop - collect_start > 0
        and    acquisition_real_sec > 3600;
```

MINIMUM	MAXIMUM	AVERAGE	STDDEV
.43	3.08	1.37	.76

```
SQL>
```

SEARCHING FOR SPECIFIC PEAKS

During the spectral review process, if analysts observe a peak at a given energy deemed important to the IDC, they should determine whether any other spectra in the database indicate a peak at the same energy location. Query (107) indicates all spectra that contain a peak between 660 and 663 keV and were collected on or after 01 August 1998. (The output generated by Query (107) has been modified to fit on the page.)

```
(107)  SQL> clear breaks
        breaks cleared

        SQL> break on  " DETECTOR"  skip 1

        SQL> select substr((gards_sample_data.site_det_code),1,10)  " DETECTOR",
                   substr((gards_sample_data.sample_id),1,6) "SAMPLE#",
                   substr(to_char ((collect_stop),'YYYY-MM-DD HH24:MI'),1,16)
                   " COLLECTION STOP",
                   to_char ((gards_peaks.energy), '9999.00') " ENERGY",
                   to_char ((gards_peaks.centroid), '9999.00') " CHANNEL",
                   to_char ((fwhm), '9.00') " FWHM",
                   to_char((area), '99999999.00') " NET AREA",
                   to_char(((area_err / area) * (100)), '9999.00') " %RELERR"
        from      guards_sample_data, guards_peaks
        where     guards_sample_data.sample_id= guards_peaks.sample_id
        and       guards_sample_data.data_type = 'S'
        and       guards_sample_data.spectral_qualifier = 'FULL'
        and       guards_sample_data.acquisition_real_sec > 3600
        and       guards_peaks.energy >= 660
        and       guards_peaks.energy <= 663
        and       guards_sample_data.collect_stop > '01-Aug-98'
        order by  site_det_code, collect_stop, acquisition_start;
```

DETECTOR	SAMPLE	COLLECTION STOP	ENERGY	CHANNEL	FWHM	NET AREA	%RELERR
DE002IAR3	40267	1998-08-03 08:24	661.51	1292.28	3.13	299.57	22.19
FI001F09	40078	1998-08-03 05:32	661.68	1986.36	1.25	542.03	11.49
	40230	1998-08-10 06:51	661.61	1986.07	1.30	334.91	12.05
US001USA1	40212	1998-08-14 20:32	661.85	1356.34	1.10	92.43	58.74

```
SQL>
```


References

- [And90b] Anderson, J., and Swanger, H., *CSS Version 3 Database: SQL Tutorial*, Science Applications International Corporation, SAIC-90/1437, 1990.
- [ANS86] American National Standards Institute, *Database Language SQL*, Document ANSI X3.135-1986, 1986.
- [Cod90] Codd, E. F., *The Relational Model for Database Management: Version 2*, Addison-Wesley Publishing Company, Reading, MA, 1990.
- [Dat86] Date, C. J., *Relational Database Selected Writings*, Addison-Wesley Publishing Company, Reading, MA, 1986.
- [Eme89] Emerson, S. L., Darnovsky, M., and Bowman, J. S., *The Practical SQL Handbook*, Addison-Wesley Publishing Company, Reading, MA, 1989.
- [Fle89] Fleming, C. C., and von Halle, B., *Handbook of Relational Database Design*, Addison-Wesley Publishing Company, Reading, MA, 1989.
- [Gil89] Gillaspay, J. L., "Oracle SQL Query Strategies," *Database Programming & Design*, Vol. 2, No. 3, pp. 50-53, 1989.
- [Gru90] Gruber, M., *Understanding SQL*, SYBEX, Inc., Alameda, CA, 1990.
- [Hur88] Hursch, C. J., and Hursch, J. L., *SQL, The Structured Query Language*, TAB Books, Inc., Blue Ridge Summit, PA, 1988.

References ▼

- [IDC5.1.1Rev1] Science Applications International Corporation, Pacific-Sierra Research Corporation, *Database Schema (Part 1, Part 2, and Part 3), Revision 1*, SAIC-99/3009, PSR-99/TN1142, 1999.
- [Koc97] Koch, G., and Loney, K., *Oracle8: The Complete Reference*, Oracle Press, Osborne/McGraw-Hill, Berkeley, 1997.
- [Lus88] Lusardi, F., *The Database Experts' Guide to SQL*, McGraw-Hill, Inc., New York, 1988.
- [Ora88] Oracle Corporation, *SQL Language Reference Manual*, 1988.
- [Sch88] Schweighofer, R., "High Performance: Choosing the Optimum Access Strategy," *Oracle International User Week*, Database Topics 6:1-11, 1988.
- [van88a] van der Lans, R. F., *Introduction to SQL*, Addison-Wesley Publishing Company, Reading, MA, 1988.
- [van88b] van der Lans, R. F., *The SQL Standard: A Complete Reference*, Prentice Hall International (UK) Ltd, Hertfordshire, England, 1988.

Glossary

A

ANSI

American National Standards Institute

C

CMR

Center for Monitoring Research.

D

DB

Database.

DBA

Database Administrator.

DBMS

Database Management System.

E

epoch time

Number of seconds after January 1, 1970 00:00:00.0.

G

GMT

Greenwich Mean Time.

GSETT

Group of Scientific Experts Technical Test.

I

IDC

International Data Centre.

ISO

International Standards Organization.

O

Oracle

Vendor of PIDC and IDC database management system.

P

PIDC

Prototype International Data Centre.

Glossary ▼**R****RDBMS**

Relational Database Management System.

S**SAIC**

Science Applications International Corporation.

SQL

Structured Query Language; a language for manipulating data in a relational database.

T**time, epoch**

See epoch time.

U**UNIX**

Trade name of the operating system used by the Sun workstations.

UTC

Universal Coordinated Time.

W**workstation**

High-end, powerful desktop computer preferred for graphics and usually networked.

Index

Symbols

! 11
% 12
/ 49
< 15, 18
<= 15, 18
= 11, 18
> 18
>= 16, 18
_ 12

A

all_catalog 81, 82
AND 13
ANSI 2
APPEND 50
AS, (with CREATE TABLE) 40
asc 19
attribute. See column.

B

BETWEEN 16, 18

C

Cartesian product vi, 30
CHANGE 49
CLEAR BUFFER 54
COLUMN 49
column 4
COMMIT 44, 92
concentration ranges 102
CONNECT 8
constraints 40
correlation name vi, 27
 restrictions 28
CREATE TABLE 39

D

database instance vi
datatypes 40
DELETE FROM 43
DESC 19
DESCRIBE 83
DISTINCT 21
DROP TABLE 41
dual 62
duplicate rows 21

E

EDIT 51
epoch time 63
etoh 64
EXISTS 32, 34
EXIT 9

Index ▼

F

field. See column.
FORMAT 49
FROM 9

G

GROUP BY 22
group functions 22

H

HAVING 24
HOST 52

I

IN 18, 31
INSERT INTO 41

J

join vi, 25, 29
 natural vi, 25, 32
 outer vi, 35, 38

K

key vi, 4
 alternate
 foreign
 primary

L

LIKE 12, 18

LIST 49
login.sql 54

N

nested query 31
NOT 13
 with EXISTS 35, 37
 with IN 18
nuclide peaks 103

O

one-to-many 35
one-to-one-or-none 36
OR 13
ORDER BY 18, 20

P

password 8
predicate vi, 11

Q

query buffer 48

R

record. See row.
relational database 2, 4
ROLLBACK 44
row vii, 4
RUN 49

S

- SAVE 52
- SELECT 9
- sequence numbers 20
- specific nuclide search 101
- SPOOL 55
- SQL*Plus extensions 47
- SQLPATH 54
- sqlplus 8
- START 52
- subquery vi, 33

T

- table 4
- TO_CHAR() 60
- TO_DATE() 60
- transaction vii
- tuple. See row.

U

- UNION 38
- UPDATE 43
- user_catalog 82

V

- VALUES, (with INSERT) 41
- view vii

W

- WHERE 10

